

A w a y 3 D 3 . 6 E s s e n t i a l s

三维动漫 游戏开发 编程入门

Matthew Casperson 著 / 胡新荣 编译



清华大学出版社

关于Away3D编程

Away3D是Flash平台最流行的实时3D引擎之一。可以利用它的特效、集成第三方文件库等技术创建复杂的3D环境和3D动画场景。使用这个引擎的可能性是无限的，但是你能充分利用它的所有特性来建立一个完美的3D应用吗？

这是一本引导你学习Away3D的最佳著作。利用这本实战指南，你就可以创建基本3D对象，显示逼真的动画人物，构建细节非常复杂的3D场景等。本书中的许多技巧可以帮助你避免常见的陷阱，从正确地显示你的第一个球体到建立一个完整的3D城市，直至建立高度逼真的3D Flash应用程序。

这本著作适合于那些希望在Flash中使用Away3D引擎创建3D应用的初学者，也适合于经验丰富的Flash开发人员。无论你是Away3D的初学者还是经验丰富的程序员，本书都为你提供了将Flash带入下一个维度的坚实基础。该书也可以作为那些已经很熟悉Away3D的Flash开发人员的参考书。

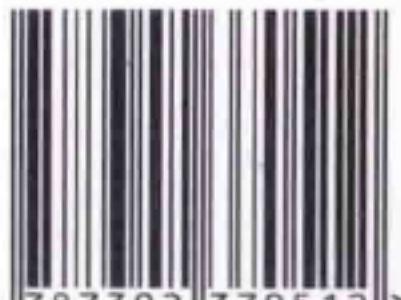
从本书中你可以学到：

- 创建效果惊人的、具有复杂纹理的3D环境
- 包括3D文本在内的各种3D对象的动画和变换
- 无须昂贵的硬件，利用Away3D的优化技巧而不影响视觉效果
- 绘制一些基本形状（如立方体、圆锥体、球体和平面体）的时候无须从基本元素开始构建它们
- 在你的3D应用中加入引人注目的特殊效果
- 按照你的意愿来对3D文本进行卷绕、曲线化、修改和弯曲
- 从各个角度聚焦相机，观察3D对象
- 使用精灵(sprites)和精灵类

清华大学出版社数字出版网站

WQBook  
www.wqbook.com

ISBN 978-7-302-37051-2



9 787302 370512 >

定价：59.00元

Away3D 3.6 Essentials

三维动漫 游戏开发 编程入门

Matthew Casperson 著 / 胡新荣 编译

清华大学出版社

Copyright © Packt Publishing 2011.

First published in the English language under the title "Away3D 3.6 Essentials"

北京市版权局著作权合同登记号 图字：01-2014-2208

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

三维动漫游戏开发编程入门/(美)卡斯帕森(Casperson, M.)著;胡新荣编译.--北京:清华大学出版社,2014

书名原文: Away3D 3.6 essentials

ISBN 978-7-302-37051-2

I. ①三… II. ①卡… ②胡… III. ①三维—动画—游戏程序—程序设计 IV. ①TP391.41

中国版本图书馆 CIP 数据核字(2014)第 143082 号

责任编辑:梁颖 薛阳

封面设计:傅瑞学

责任校对:焦丽丽

责任印制:王静怡

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座 邮 编:100084

社总机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

课 件 下 载: <http://www.tup.com.cn>, 010-62795954

印 刷 者:清华大学印刷厂

装 订 者:三河市新茂装订有限公司

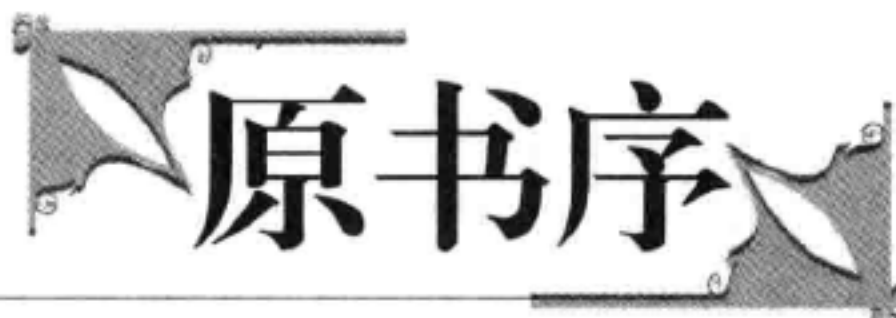
经 销:全国新华书店

开 本:185mm×230mm 印 张:19.25 字 数:422 千字

版 次:2014 年 9 月第 1 版 印 次:2014 年 9 月第 1 次印刷

定 价:59.00 元

产品编号:059314-01



Away3D 是 Flash 可用的最流行的、最受开发者欢迎的实时 3D 引擎,它可创建多种 3D 应用程序,包括可视的精细 3D 环境,显示 3D 动画模型,建立 3D 文本,还有大量的 3D 特殊效果显示。利用 Away3D 和少量的 ActionScript 代码,给开发者提供了无穷的创意空间。

本书将引导你学习 Away3D 中各种可用的功能,这些特性对 Flash 平台是完全开放的,通过书中的实例和一些实用的技巧,你可以很快建立并运行一个用 Away3D 编写的应用程序。

本书从最基本的要素开始,通过下载 Away3D 的源代码,教会你在很多程序编写工具如 Flex Builder,Flash Builder,FlashDevelop 和 Flash CS4 里使用 Away3D 来创建你的第一个 Away3D 的应用程序。接下来,你可以很容易地从零开始创建第一个原始 3D 对象,然后,逐步教你建立非常好的、具有精细纹理和动画效果的 3D 环境。你还将学会如何使用应用程序响应用户的请求,学会使照相机聚焦的方法,以及从任意的视角观察场景。同时,还会教你很多的优化技术,使应用程序获得在 Away3D 里有最好的运行性能,而不会降低视觉效果,使应用程序提升到一个新的级别。

本书从显示一个球开始,到创建一个完整的 3D 城市,逐步教你必须掌握的编程步骤、编程技术和大量的技巧,使你避免犯常见错误。

本书包含的内容:

第 1 章,搭建第一个 Away3D 应用程序,教你如何在各种不同的集成开发环境 (IDEs、Flex Builder、Flash Builder、FlashDevelop 和 FlashCS4) 里,搭建第一个 Away3D 的应用程序。

第 2 章,建立并显示原始 3D 模型,在这一章你将浏览 Away3D 里的各种有用的原始对象。

第 3 章,移动对象,教你如何在场景里直接移动、旋转、缩放 3D 对象,或者通过缓动库来进行。

第 4 章,Z-排序,探索能用的景深排序技巧以及在 Away3D 程序里出现的渲染问题。

第 5 章,材质,带你进入到 Away3D 里,查看它包含的各种材质,从显示单色的材质到 Pixel Bender 的高级材质,此外本章还将讲述光源的知识。

第 6 章,模型和动画,你将学会如何载入和显示用其他 3D 建模软件创建的静态 3D 模型及其动画效果。

第 7 章,照相机,显示那些影响场景效果的各个属性,展示 Away3D 里的各种照相机类,它们使你能够追踪和查看场景里的 3D 对象。

第 8 章,鼠标交互,你会学到如何响应鼠标的请求,建立交互的应用程序。

第 9 章,使用精灵的特殊效果,本章展示了各种各样的特殊效果,包括如何与 Stardust 粒子引擎集成。

第 10 章,建立 3D 文本,本章将讲述如何创建和管理 3D 文本。

第 11 章,突显法和修改工具,本章将讲述如何直接使用 Away3D 创建复杂的 3D 对象,而不使用其他 3D 建模软件。

第 12 章,过滤器和后继处理效果,本章讲述如何在 Away3D 应用程序里添加振奋人心的效果。

第 13 章,Away3D 的运行技巧,教你如何优化 Away3D 应用程序,使你能创建维持高级别的运行环境。

这本书可以为你提供:

希望建立能参与性与交互性强、引人注目的网站或吸引人的 3D 游戏的人都会欣赏 Away3D 的强大功能,本书提供了如何利用这些功能的所有资料和信息。你需要做的工作是通过互联网来下载 Away3D 和诸如 Flex/Flash Builder、Flash CS4 或者 Flash Develop 这样的 ActionScript 集成开发环境,这些软件都是免费下载使用的。

这本书面向的对象:

这本书是为那些希望在 Flash 中使用 Away3D 引擎创建 3D 应用的初学者和经验丰富的 Flash 开发人员而写的。无论你是 Away3D 的初次使用者还是经验丰富的开发者,这本书都将为你进一步学习 Flash 提供一个坚实的基础,同时这本书也可以作为已经熟悉 Away3D 的 Flash 开发人员的参考指南。

相关约定:

在本书中你可以发现很多风格的文本区分不同类型的信息,下面列举了一些文本风格及其含义的解释。

文本中的代码词显示如下:

“通过扩展 Away3Dtemplate 类,我们可以用 SphereDemo 类和少量的几行代码就可以创建一个简单的 3D 应用程序。”

代码段建立如下:

```
import away3d.core.base.Object3D;
```



```
import away3d.primitives.Cone;  
import away3d.primitives.Cube;  
import away3d.primitives.Cylinder;  
import away3d.primitives.GeodesicSphere;
```

新的术语和重要的词都加粗显示了。例如在屏幕、菜单或对话框里显示的文本：“在 Flex/Flash Bulider 和 Flash CS4 中将 TweenLite 库添加到 Source path, 或者在 FlashDevelop 中添加到 Project Classpaths.”

读者反馈意见：

本书深受读者欢迎,让我们了解你对本书的看法——无论是喜欢本书还是不喜欢本书。读者的反馈意见是非常重要的,因为这样有助于我们对读者真正需要的主题进行改进。

请把反馈意见发到电子邮箱: feedback@packtpub.com,并在邮件的主题中注明书名。

如果您有需要的书并希望我们出版,请在网站 <http://www.packtpub.com> 的 SUGGEST A TITLE 中给予提醒或者给我们发邮件: suggest@packtpub.com。

如果有你专业领域的主题,你有兴趣出版或参编相关的书,也可以通过 <http://www.packtpub.com/authors> 查看作者指南。

消费者支持：

现在你是 Packt 出版社的尊贵主人,我们为你的书籍消费提供了最大帮助。

下载本书的例程代码

你可以通过你的账号从 <http://www.packtpub.com> 购买 Packt 出版社的书籍并下载相应的例程源码。如果你在别处购买的 Packt 出版社的书籍,可以访问 <http://www.packtpub.com/support> 并注册,我们将把例程源码文件直接发电子邮件给你。

错误之处：

尽管我们十分仔细认真以保证我们内容的正确性,但错误还是难以避免。如果你在我們的書中發現一個錯誤——無論是文本錯誤還是代碼錯誤——如果你願意告知我們,我們將十分感激。這樣,你可以幫助其他讀者,也有助於幫助我們改進這本書的後續版本。如果你發現書中的任何錯誤,可以訪問 <http://www.packtpub.com/support>,選擇該書,單擊鏈接“errata submission form”,輸入錯誤的詳細描述,報告這些錯誤。一旦你提交的錯誤通過審核,你的提交將被接受並上傳到我們的網站,或被增加到 Errata 的已有錯誤列表中。從 <http://www.packtpub.com/support> 可以查看你選中書目的所有錯誤。

版权声明：

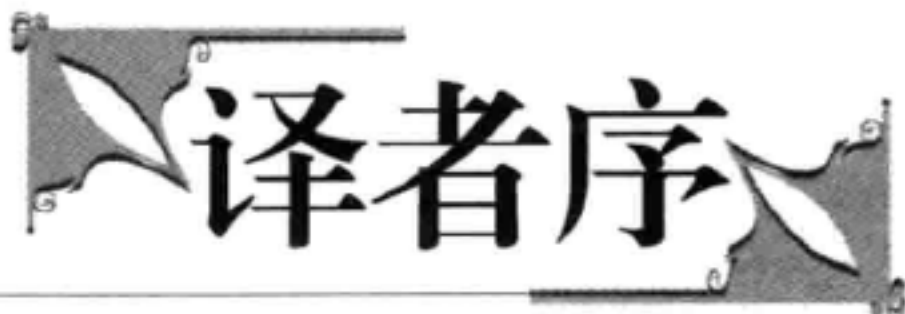
对所有媒体而言,互联网上的版权材料的盗版是一个持续存在的问题。在 Packt,我们非常认真地对待版权和著作权的保护。如果你在互联网上遇到以任何形式非法拷贝我们的著作,请立即为我们提供地址或网站,这样我们可以采取补救措施。

请将可疑盗版资料的地址发给我们: copyright@packtpub.com。

我们十分感激你为保护我们作者所提供的帮助,我们将尽力为你提供有价值的内容。

问题：

如果你对本书的任何内容有疑问的话请联系我们: question@packtpub.com,我们将尽力解决这个问题。



当前,国内在很多地方投入了大量的人力、物力和财力,开发基于互联网的三维动漫游戏、工业产品展示、实时模拟教学等。然而却缺少一本专业的、教读者如何编写开发三维动漫程序的书。为此作者编译了本书,以期推动 3D 动漫的教学、研究和开发。

读者在阅读本书之前,最好学习过 Java, MyEclipse, Flash, 3D Max 和“计算机图形学”这些前期课程。没有学也没关系,跟着本书的讲解,照着例子练习,一样可以学会 Away3D 编程。

在此书的编译过程中,得到了实践经验丰富的黄传贤教授、伍宁工程师的精心指正和帮助,邓盾同学也协助编者对书中的部分例程进行编排和调试,本书的出版受到了国家自然科学基金项目(61103085)资助,并由武汉纺织大学学术著作出版基金资助出版,在此一并感谢。

本书各章讲解的所有例子均在 Flash Builder 4.0 和 Away3D 4.0 的环境下调试且成功运行过。

由于编译者水平所限,书中内容难免有不对之处,敬希读者指教,以便再版时改正。

胡新荣

2014 年 7 月

第 1 章 搭建第一个 Away3D 应用程序	1
1.1 在 Flash Player 10 和 Flash Player 9 之间选择适合它的 Away3D 版本	1
1.2 下载 Away3D	2
1.3 下载 Away3D 的源码压缩 ZIP 文件	2
1.4 使用版本管理 SVN 下载 Away3D 的源码	3
1.5 建立空的 Away3D 工程项	4
1.5.1 Adobe Flex Builder 或 Flash Builder	4
1.5.2 FlashDevelop	4
1.5.3 Adobe Flash CS4	5
1.6 配置针对 Flash Player 10 的运行环境	6
1.6.1 Adobe Flex Builder 和 Adobe Flash Builder	7
1.6.2 FlashDevelop	9
1.6.3 Adobe Flash CS4	10
1.7 建立初始的应用程序	10
1.8 运行 Away3DTemplate	13
1.8.1 Adobe Flex Builder 和 Adobe Flash Builder	13
1.8.2 FlashDevelop	14
1.8.3 Adobe Flash CS4	15
1.8.4 最终结果	16
1.9 在一个 3D 场景中定位对象	16
1.10 继承 Away3DTemplate 类构建一个场景	17
1.10.1 运行 SphereDemo 应用程序	18
1.10.2 最终结果	19
第 2 章 建立并显示原始 3D 模型	20
2.1 一个 3D 对象的基本元素	20

2.1.1	顶点	21
2.1.2	三角面	21
2.1.3	Sprite3D 精灵	23
2.1.4	段	25
2.2	UV 纹理贴图坐标系统	25
2.3	创建原始的 3D 对象	26
第 3 章	移动对象	50
3.1	全局坐标、父坐标和局部坐标	50
3.1.1	世界空间	50
3.1.2	父空间	51
3.1.3	局部空间	53
3.2	转换函数/属性及其坐标系统	55
3.3	修改位置	57
3.3.1	x、y 和 z 属性	57
3.3.2	位置属性	57
3.3.3	移动函数	58
3.3.4	moveTo() 函数	59
3.3.5	translate() 函数	59
3.3.6	修改旋转	59
3.3.7	修改缩放	64
3.3.8	修改转换	65
3.3.9	渐变操作	65
3.4	嵌套	68
第 4 章	景深排序	72
4.1	画家算法	72
4.2	场景排序	73
4.3	调整排序的顺序	75
4.3.1	前推和后推属性	75
4.3.2	屏幕 Z 位移属性	76
4.3.3	画布属性	77
4.4	有关 Z-排序的说明	79
4.5	附加的渲染器	79

第 5 章 材质	86
5.1 纹理和材质的区别	86
5.2 资源管理	87
5.3 在 Away3D 中定义彩色	88
5.3.1 用整数定义色彩	88
5.3.2 用颜色名字的字符串定义色彩	88
5.4 Pixel Bender	89
5.5 光源和材质	90
5.6 着色技术	91
5.6.1 纹理映射	91
5.6.2 法向贴图	91
5.6.3 环境映射	93
5.6.4 平面着色	94
5.6.5 Phong 着色	94
5.7 应用材质	94
5.8 基本材质	104
5.8.1 线色彩材质	104
5.8.2 线框材质	105
5.8.3 彩色材质	106
5.8.4 Bitmap 材质	107
5.8.5 动画材质	110
5.8.6 复合材质	113
5.8.7 光源材质	117
5.8.8 Pixel Bender 材质	123
5.8.9 从外部文件载入纹理图	130
第 6 章 模型和动画	133
6.1 Away3D 支持的 3D 格式	134
6.2 输出 3D 模型	134
6.2.1 从 3ds Max 输出模型文件	134
6.2.2 从 MilkShape 输出模型文件	136
6.2.3 从 Sketch-up 输出模型	136
6.2.4 从 Blender 输出模型	137
6.2.5 有关 Collada 输出器的注意事项	138

6.3	载入 3D 模型	139
6.4	动画模型	139
6.4.1	MD2 载入一个嵌入式文件	140
6.4.2	MD2 载入一个外部文件	142
6.4.3	Collada 载入一个嵌入式文件	144
6.4.4	Collada 载入一个外部文件	146
6.4.5	AS 载入转换后的模型	147
6.5	静态模型	148
6.5.1	3DS 载入嵌入文件	149
6.5.2	3DS 载入外部文件	150
6.5.3	AWD 载入嵌入文件	151
6.5.4	AWD 载入外部文件	153
6.5.5	ASE 载入嵌入文件	154
6.5.6	ASE 载入外部文件	155
6.5.7	OBJ 载入嵌入文件	156
6.5.8	OBJ 载入外部文件	158
6.5.9	初始化对象的使用问题	159
6.6	把载入的模型转换成 ActionScript 类	161
第 7 章	照相机	163
7.1	照相机类的属性	163
7.2	照相机的镜头	164
7.2.1	放大焦距镜头和透视镜镜头类	165
7.2.2	球面镜头类	165
7.2.3	正交镜头类	166
7.3	照相机类	167
7.3.1	目标照相机	173
7.3.2	旋转照相机	174
7.3.3	跟踪照相机	176
第 8 章	鼠标互动性	177
8.1	Away3D 鼠标事件	177
8.2	ROLL_OVER/ ROLL_OUT 和 MOUSE_OVER/ MOUSE_OUT 之间的区别	179
8.3	将鼠标的位置投影到场景	183

第 9 章 使用精灵的特殊效果	191
9.1 使用 Sprite3D 类	191
9.2 使用定向精灵类	195
9.3 使用景深精灵类	199
9.4 使用粒子系统	203
9.4.1 建立 Away3D Stardust 初始化程序类	203
9.4.2 建立 Away3D 星团粒子渲染器	205
9.4.3 建立星团发射器	207
9.4.4 把上面的全部功能集合到一起	211
第 10 章 建立 3D 文本	213
10.1 嵌入文字	214
10.1.1 在场景里显示文本	214
10.1.2 3D 文本材质	217
10.2 突出显示 3D 文本	217
10.3 弯曲 3D 文本	219
第 11 章 突显法和修改工具	227
11.1 使用 PathExtrusion 类建立标记	227
11.2 使用 LinearExtrusion 类建立围墙	230
11.3 使用 LatheExtrusion 类建立花瓶	232
11.4 使用 SkinExtrusion 类建立地形图	235
11.5 用海拔阅读类读出地形图表面高度	239
11.6 高度映射修改类	243
第 12 章 过滤器和后继处理效果	247
12.1 Flash 和 Away3D 过滤器	247
12.1.1 Flash 过滤器	248
12.1.2 Away3D 的过滤器	257
12.2 渲染器会话对象	260
第 13 章 Away3D 的运行技巧	266
13.1 确定当前的帧速度	266
13.2 设置最大的帧速度	268

13.3	设置 Flash 舞台的质量低一点	268
13.4	减小视口尺寸的大小	270
13.5	缩放视口输出	271
13.6	三角缓存	271
13.7	模型的细节层次	275
13.8	Away3D 过滤器	277
13.8.1	Z 景深过滤器	277
13.8.2	最大多边形过滤器	278
13.8.3	在后台进行画图	278
13.9	模型格式	287

搭建第一个Away3D应用程序

搭建第一个 Away3D 应用程序不是一件容易的事情,因为在能够编写出一行代码之前,还要完成很多必要的步骤。

本章将通过这些必要步骤,构建第一个 Away3D 应用程序,并使之运行起来。

本章主要内容:

- 不同的 Away3D 版本
- 下载 Away3D
- 配置好应用程序的开发环境
- 通过使用 Away3D,明白其原理概貌
- 建立一个样本应用程序
- 使用所选择的程序编写工具,编译应用程序

1.1 在 Flash Player 10 和 Flash Player 9 之间 选择适合它的 Away3D 版本

2008 年,Adobe 发布 Flash Player 10,它带来了许多运行方面的性能改进,并且添加了许多新的性能,这给 3D 引擎,也给 Away3D 引擎带来了很大的好处。然而这些新的性能并不向后兼容,也就是说 Flash Player 10 能运行的 Flash 应用程序,在 Flash Player 9 上不能运行。为了既适合 Flash Player 10,也适合 Flash Player 9,这样就有了两个 Away3D 引擎

版本：一个是 Away3D version 2, 它适合 Flash Player 9; 另一个是 Away3D version 3, 它适合 Flash Player 10。

当 Flash Player 10 最初发布时, 开发者有充分的理由去支持老的版本 Flash Player 9。因为能安装 Flash Player 10 的机器相对来说还是很少的。目前, 能安装 Flash Player 10 的机器已超过 90%。

Flash Player 10 的 Away3D 版, 提供了许多额外的特点和性能, 给 Flash Player 10 开拓了巨大的市场。所以本书将集中讨论 Away3D 的 3.x 版。

1.2 下载 Away3D

Away3D 能够从两个网址下载, 第一个是 Away3D 官方网站 <http://Away3D.com/downloads>, 在这里可找到稳定版的 Away3D 引擎的 ZIP 压缩文件下载链接。这个版本是经过测试的, 被认为是可用的高品质的产品。

下载 Away3D 第二个网址是来自 Google 代码版本管理 SVN 的代码仓库 <http://code.google.com/p/Away3D/>。SVN 代码仓库里放的是代表当前最新的 Away3D 正式版, 使用这里的代码, 可体验到 Away3D 的最新性能。但是, Away3D 的作者还没有把他们已修改过的代码放到此正式版中来, 这里的代码仍处于测试中, 因此一般不推荐日常使用。

1.3 下载 Away3D 的源码压缩 ZIP 文件

除非另外说明, 本书中所有的例子均已用 Away3D 3.6 版编译过。请下载后按说明进行使用。在编写本书时, Away3D 3.6 是当时的最新版本, 可从 Away3D 官方网站下载 (<http://Away3D.com/downloads> 或 http://Away3D.com/download/Away3D_3_6_0.zip)。

Away3D 是一个非常活跃的项目, 每隔几个月, 可能就有新版本发布, 在阅读本书的时候, 上面说明的下载网页可能已经改变了, 不过仍然可下载新的内容。

本书的所有例子都能使用最新的 Away3D 引擎工作, 但是用 Away3D 3.6 也是保证兼容的。

一旦下载完 ZIP 文件,就可在自己的计算机上某个位置解压这些文件。但要记住解压位置,因为以后在建立自己的开发环境时,要引用这些位置。

1.4 使用版本管理 SVN 下载 Away3D 的源码

SVN 提供了一个访问最新 Away3D 版本的方法和工具。例如,TortoiseSVN 会替用户直接处理访问 Away3D 的 SVN 存储仓库的过程。

TortoiseSVN 是一个免费的、流行的、与 Windows 浏览器集成的客户端软件,它使用户很容易地访问 SVN 存储地点。可按以下的步骤下载并安装 TortoiseSVN,然后用它下载 Away3D 源代码。

- (1) 下载 TortoiseSVN(<http://tortoisesvn.net/downloads>)。
- (2) 使用默认的设置,安装 TortoiseSVN,并重新启动计算机。
- (3) 一旦重新启动计算机完成后,在硬盘文件系统方便的地方,建立文件夹,用于存放 Away3D 源代码。为了简化,可考虑在 C:/根目录下建一命名为 Away3D 的文件夹。
- (4) 右击此文件夹,并单击弹出菜单中的 SVN Checkout 选项。
- (5) 这时出现 Checkout 对话框,如图 1-1 所示。在 URL of repository 文本框中,输入“<http://away3d.googlecode.com/svn/trunk/fp10/Away3D/src>”。
- (6) 单击 OK 按钮,下载 Away3D 文件。

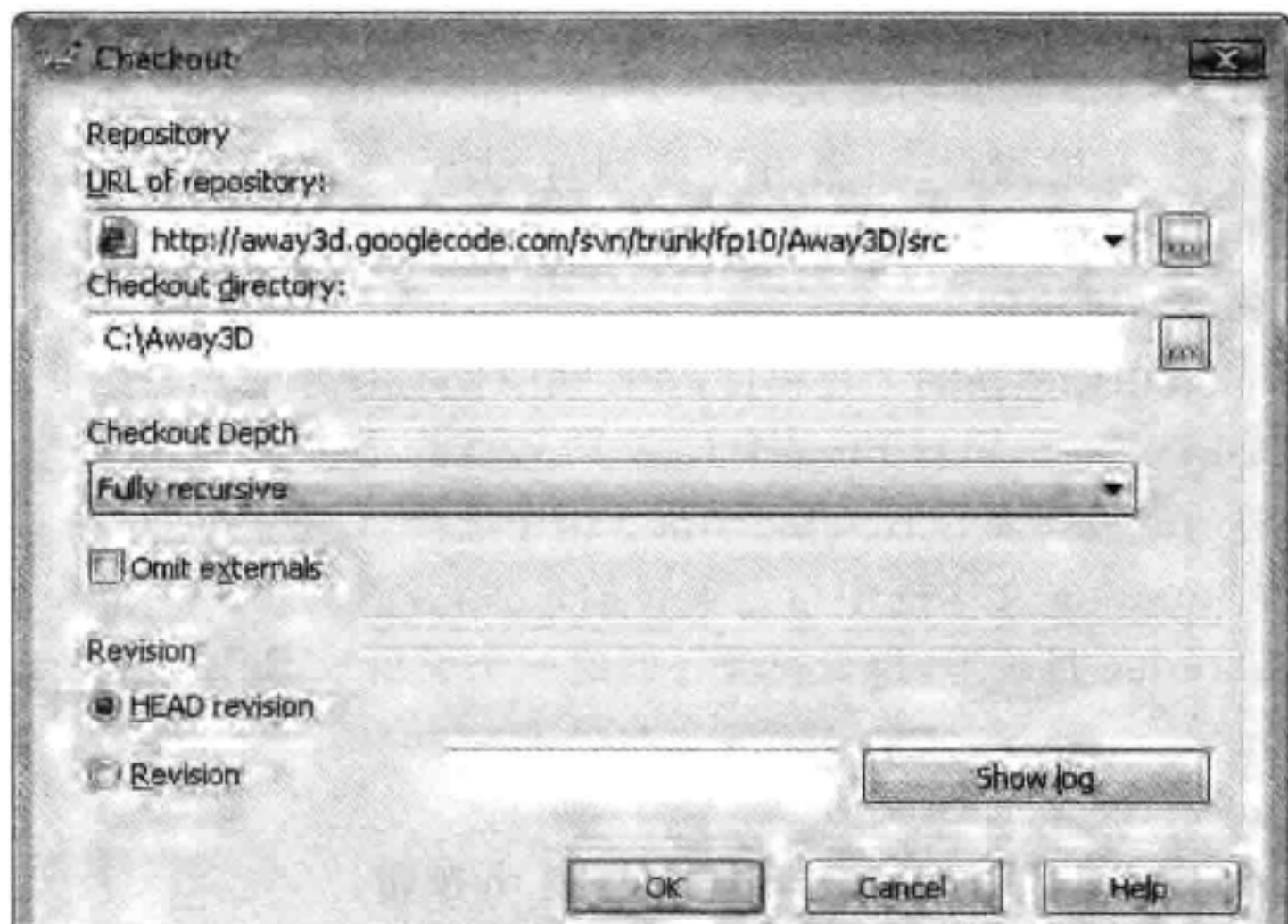


图 1-1 Checkout 对话框

1.5 建立空的 Away3D 工程项

为了能创建 Away3D 应用程序,首先需要在编写程序的工具环境中建立并配置新的工程项目。此外,为了在项目中使用 Away3D 引擎,必须对项目进行配置,以便使它能访问到 Away3D 源码。

1.5.1 Adobe Flex Builder 或 Flash Builder

在 Flash 或 Flex Builder 中创建新项目,并使项目能使用 Away3D 库的步骤如下。

- (1) 打开 Adobe Flex,并依次单击 File|New|ActionScript Project。
- (2) 在 Project name 文本框中,输入要创建项目的名字,并单击 Next 按钮。
- (3) 在 Source Path 选项卡中,单击 Add Folder 按钮。
- (4) 单击 Browse 按钮,选定保存 Away3D 源码的位置,或者在文本框中直接输入保存 Away3D 源码位置的路径。
- (5) 单击 OK 按钮。
- (6) 现在保存 Away3D 源码的文件夹就在 Source Path 中列出来了,单击 Finish 按钮,项目创建完成。

1.5.2 FlashDevelop

FlashDevelop 是一个免费的集成开发环境(Integrated Development Environment, IDE),能用于建立 Flash 应用程序,当用于与 Flex SDK 连接时(Flex SDK 也是免费的),FlashDevelop 能用来编写并编译 Flash 应用程序。FlashDevelop 可以在 <http://www.FlashDevelop.org> 网站里找到。

在 FlashDevelop 中创建新的工程项目,并使用 Away3D 库的步骤如下。

- (1) 打开 FlashDevelop,执行 Project|New Project 命令。
- (2) 从已安装的模板列表的 ActionScript 3 组中选择 AS3 Project,在 Name 文本框中,输入项目名称,在 Location 文本框中指定项目的位置,并选择 Create directory for project 复选框,这将在 Location 目录下,为工程项目创建一个子目录,如图 1-2 所示。单击 OK 按钮,创建该项目。
- (3) 执行 Project|Properties 命令。
- (4) 单击 Classpaths 标签,再单击 Add Classpath 按钮。
- (5) 浏览到保存 Away3D 源码的位置,单击 OK 按钮。
- (6) 现在 Away3D 源码的目录列在 Project Classpaths 的清单里了(注意:FlashDevelop 对 Classpaths 使用相对路径),单击 OK 按钮保存。

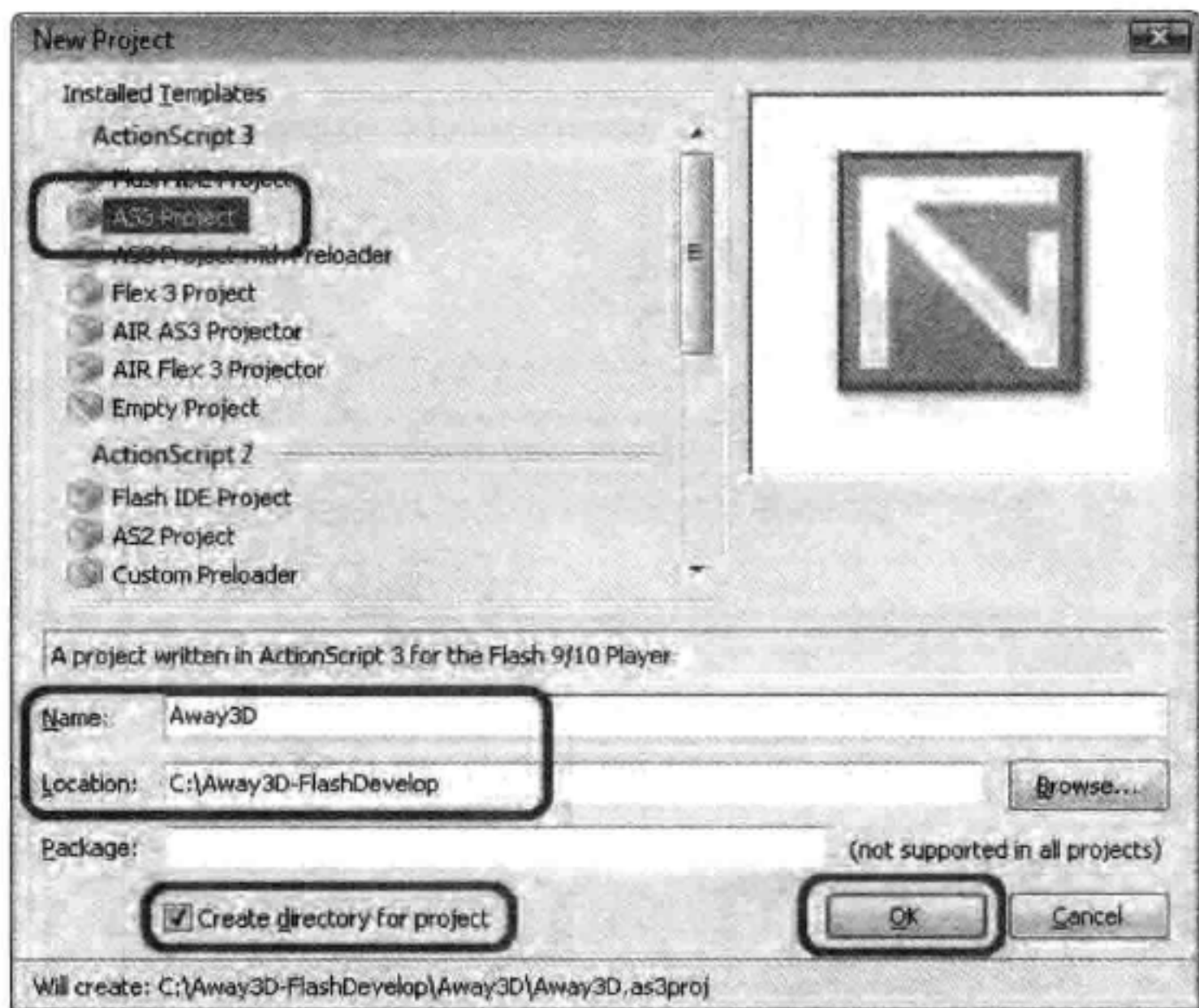


图 1-2 New Project 对话框

1.5.3 Adobe Flash CS4

Adobe Flash 是一个 Flash 编写程序的工具,Adobe Flash 在其中放置大量的着重于创建可视 Flash 动画的环境。而且,它也能使用 ActionScript 建立 Flash 应用程序。在 Adobe Flash CS4 中建立新的工程项目的步骤如下。

- (1) 打开 Adobe Flash CS4,执行 File|New 命令。
- (2) 在 General 标签中选择 Flash File(ActionScript 3.0),并单击 OK 按钮。
- (3) 执行 File|Publish Settings 命令。
- (4) 在弹出的对话框中,单击 Flash 标签,单击 Settings 按钮。
- (5) 在 Source path 选项卡下,单击 + 按钮,如图 1-3 所示。
- (6) 单击“文件夹”图标按钮,如图 1-4 所示。

(7) 由如图 1-4 所示对话框中浏览到存放有 Away3D 源码的目录,或直接在清单中输入 Away3D 源码目录的目录路径,并单击 OK 按钮,保存输入的改变。

- (8) 再单击 OK 按钮,关闭 Advanced ActionScript 3.0 Settings 对话框。
- (9) 再单击 OK 按钮,关闭 Publish Settings 对话框。
- (10) 执行 File|Save 命令,并保存 *.fla 文件到选定的目录下。

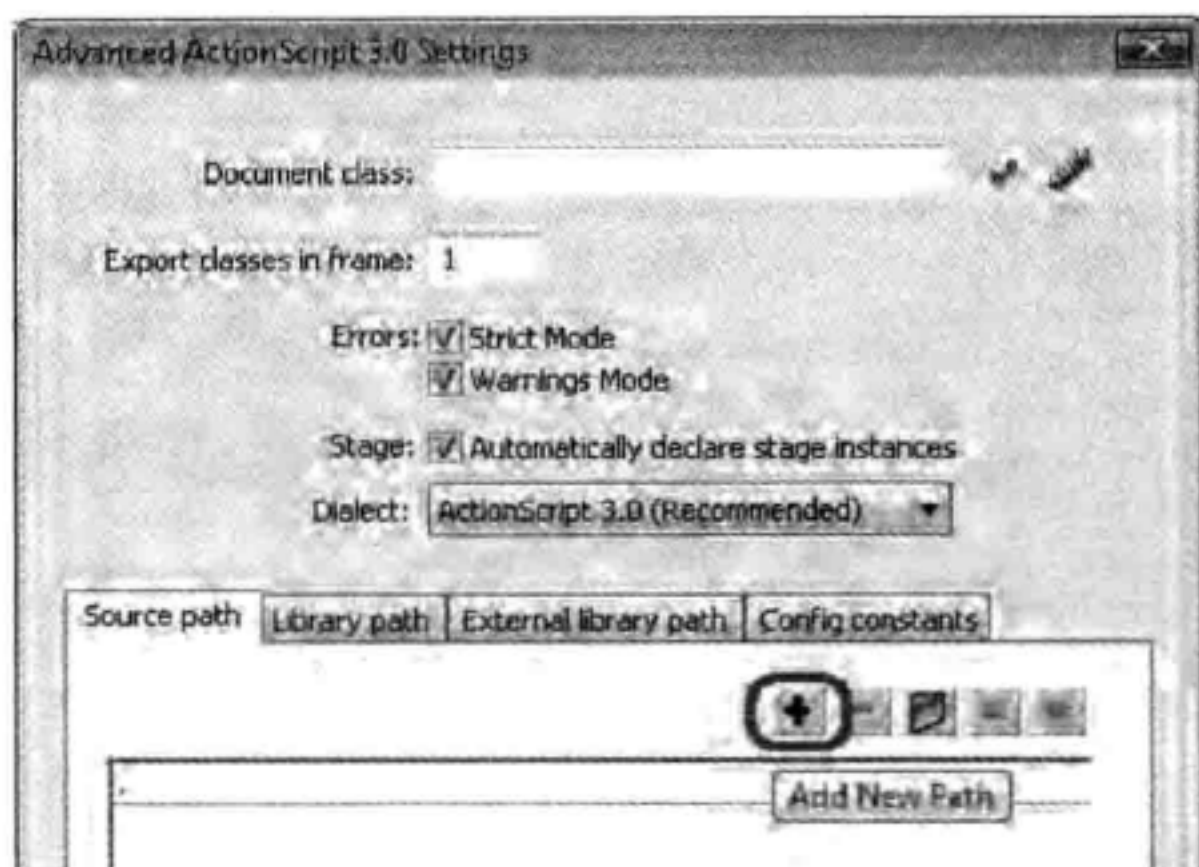


图 1-3 Advanced ActionScript 3.0 Settings 对话框 1

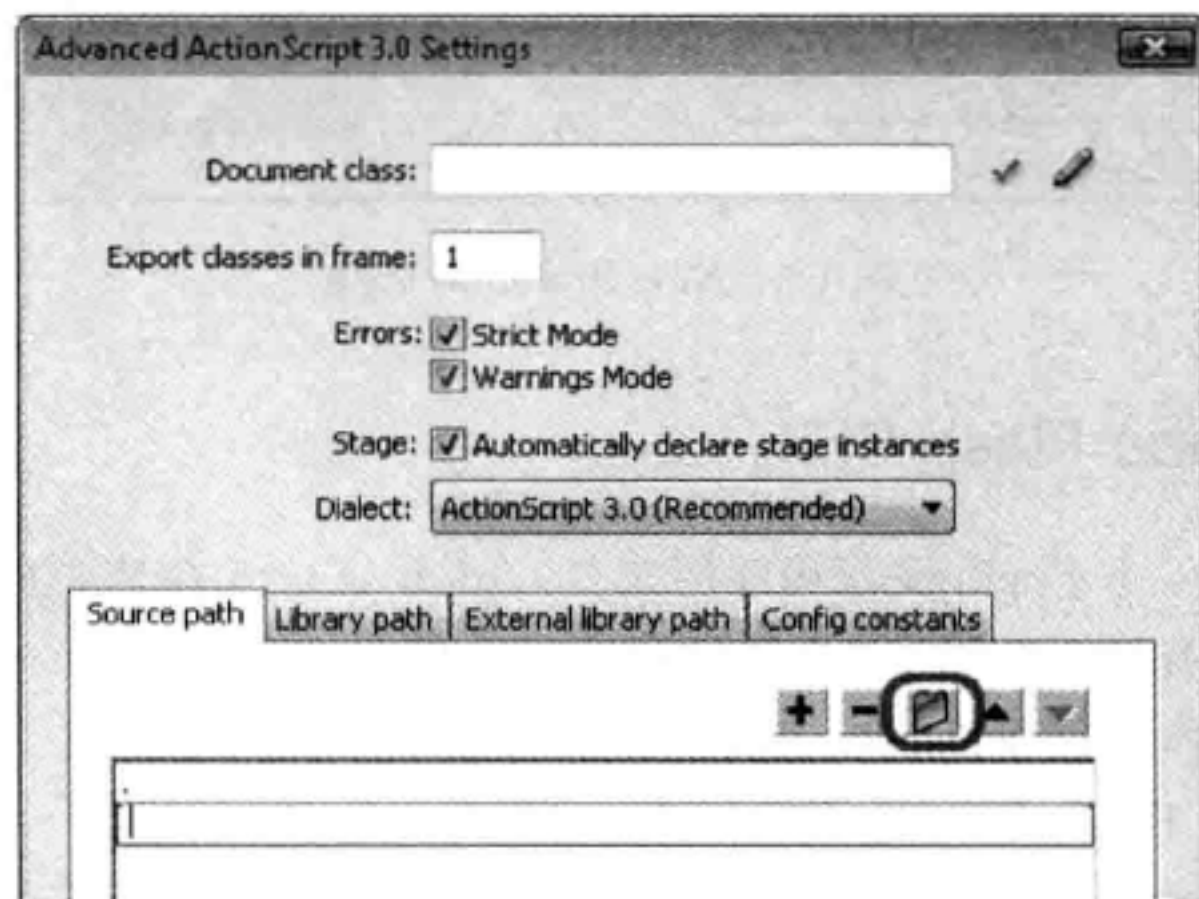


图 1-4 Advanced ActionScript 3.0 Settings 对话框 2

1.6 配置针对 Flash Player 10 的运行环境

正如本章开始所述, Away3D 3. x 是针对 Flash Player 10 的, 为了能编译使用 Away3D 3. x 应用程序, 程序编写工具必须配置成能使用 Flex SDK 3. 2 或以上版本的开发环境。

Flex SDK 可从 <http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+3> 免费下载。一旦下载完成, 选一方便的文件夹, 解压下载的 ZIP 文件。

1.6.1 Adobe Flex Builder 和 Adobe Flash Builder

以下的这些步骤,仅适用于 Flex Builder 3。在 Flash Builder 4 中,默认的 Flex SDK 版本是 4.0,默认的运行条件已经是针对 Flash Player 10 的。

(1) 执行 Project|Properties 命令,打开 ActionScript Project 项目,如图 1-5 所示。

(2) 从左边的面板中,选择 ActionScript Compiler 选项,勾选 Require Flash Player version 复选框,并输入“10.0.0”作为它的版本。然后单击 Configure Flex SDKs...链接。

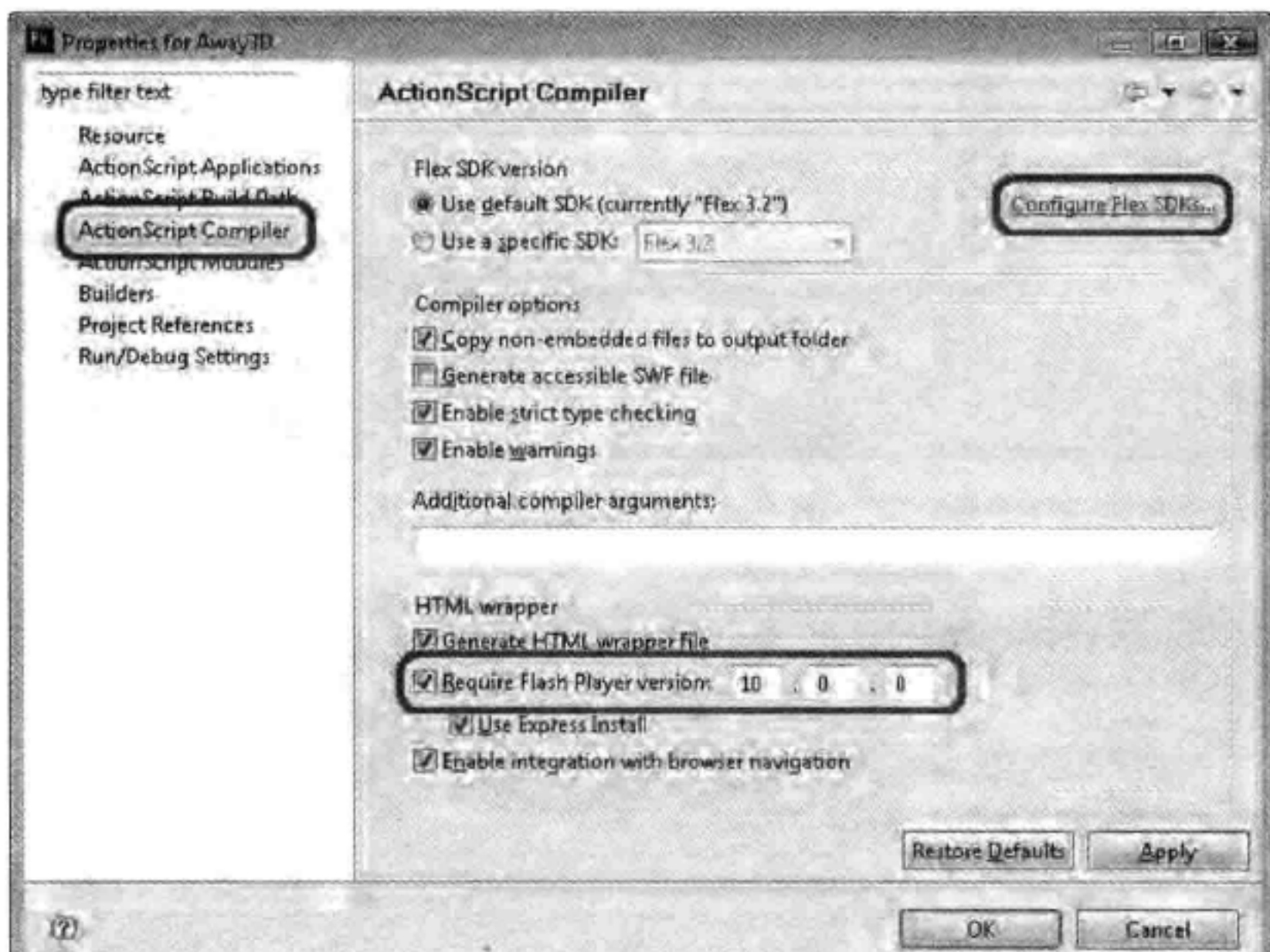


图 1-5 Properties for Away3D 窗口

(3) 单击 Add 按钮。

(4) 在 Add Flex SDK 对话框中,把 Flex SDK 解压的文件夹位置输入到 Flex SDK Location 文本框中,或单击 Browse 按钮,找到这个位置。

(5) 现在 Flex SDK name 文本框中应显示所选的 Flex SDK 版本名字。单击 Add Flex SDK 对话框中的 OK 按钮,返回如图 1-6 所示的 Preferences 窗口。

(6) 勾选 Flex SDK 名字旁边的复选框,指定 Flex 使用新的 SDK 为默认的软件,单击 OK 按钮,返回如图 1-7 所示的 Properties for Away3D 窗口。

(7) 在 Flex SDK 版本组框里,紧挨着 Use default SDK 旁显示的版本号现在应该是新的 Flex SDK 版本号,单击 OK 按钮,关闭如图 1-7 所示的 Propertis for Away3D 窗口。

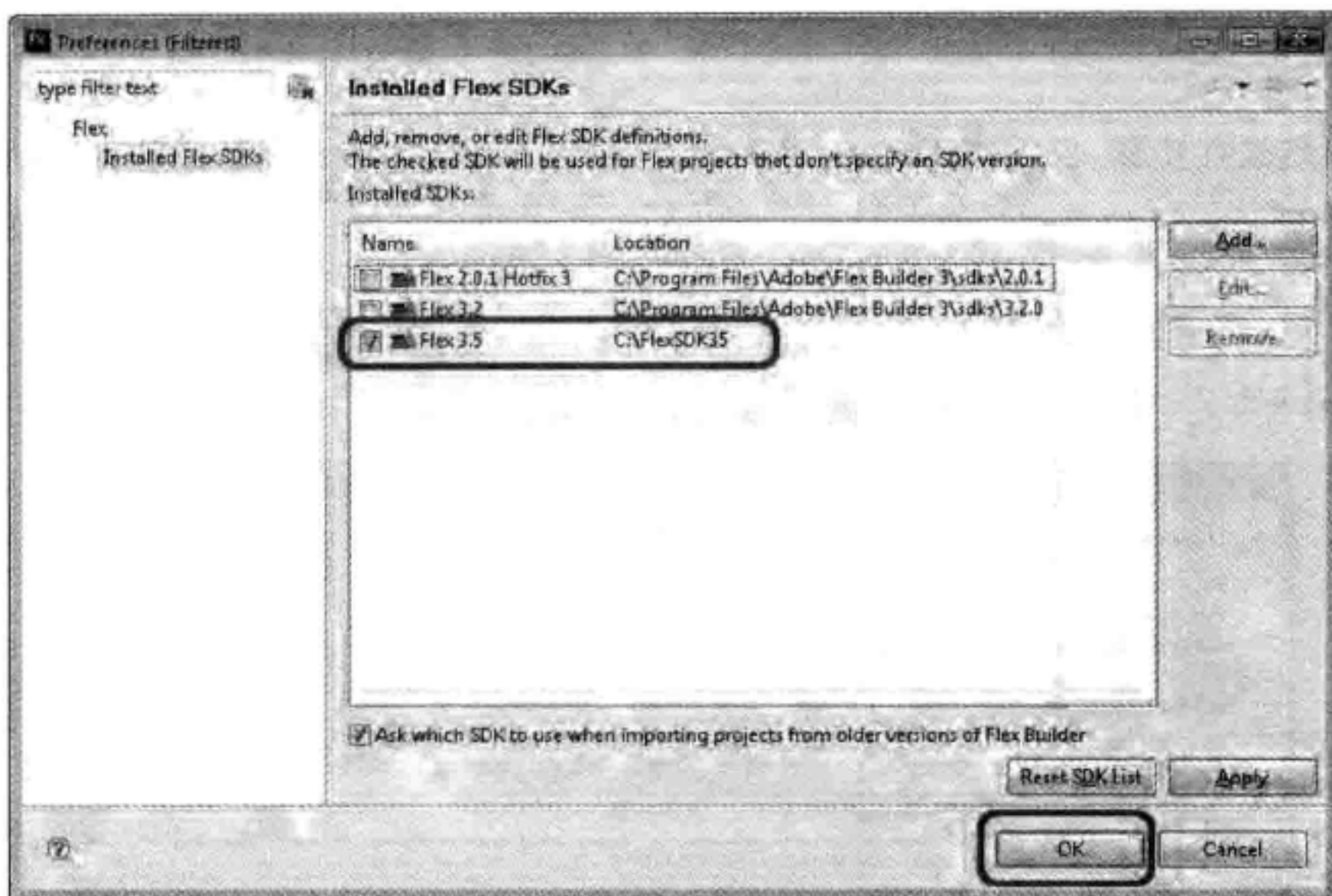


图 1-6 Preferences 窗口

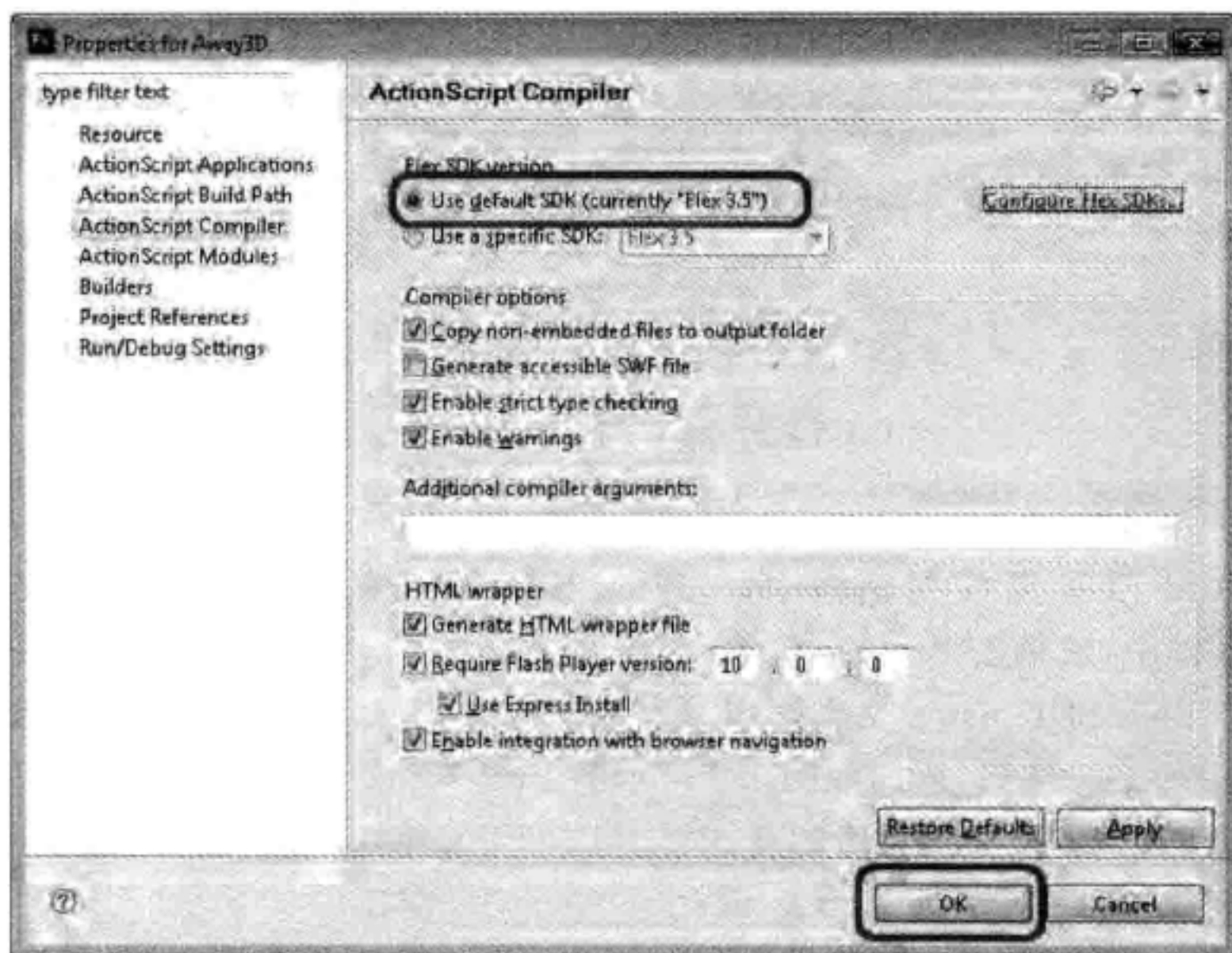


图 1-7 Properties for Away3D 窗口

1.6.2 FlashDevelop

当第一次安装 FlashDevelop 的时候,在下载 FlashDevelop 时,可以选择同时下载支持 Flash Player 10 的 Flex SDK 版本。或者按照以下的步骤手工指定适合的 Flex SDK 的位置。

(1) 执行 Tools|Program Settings 命令,打开如图 1-8 所示的 Settings 对话框。

(2) 选择左边面板中的 AS3Context 选项,并单击 Language 选项组中的 Flex SDK Location 旁最右边的带有三个小圆点的按钮。

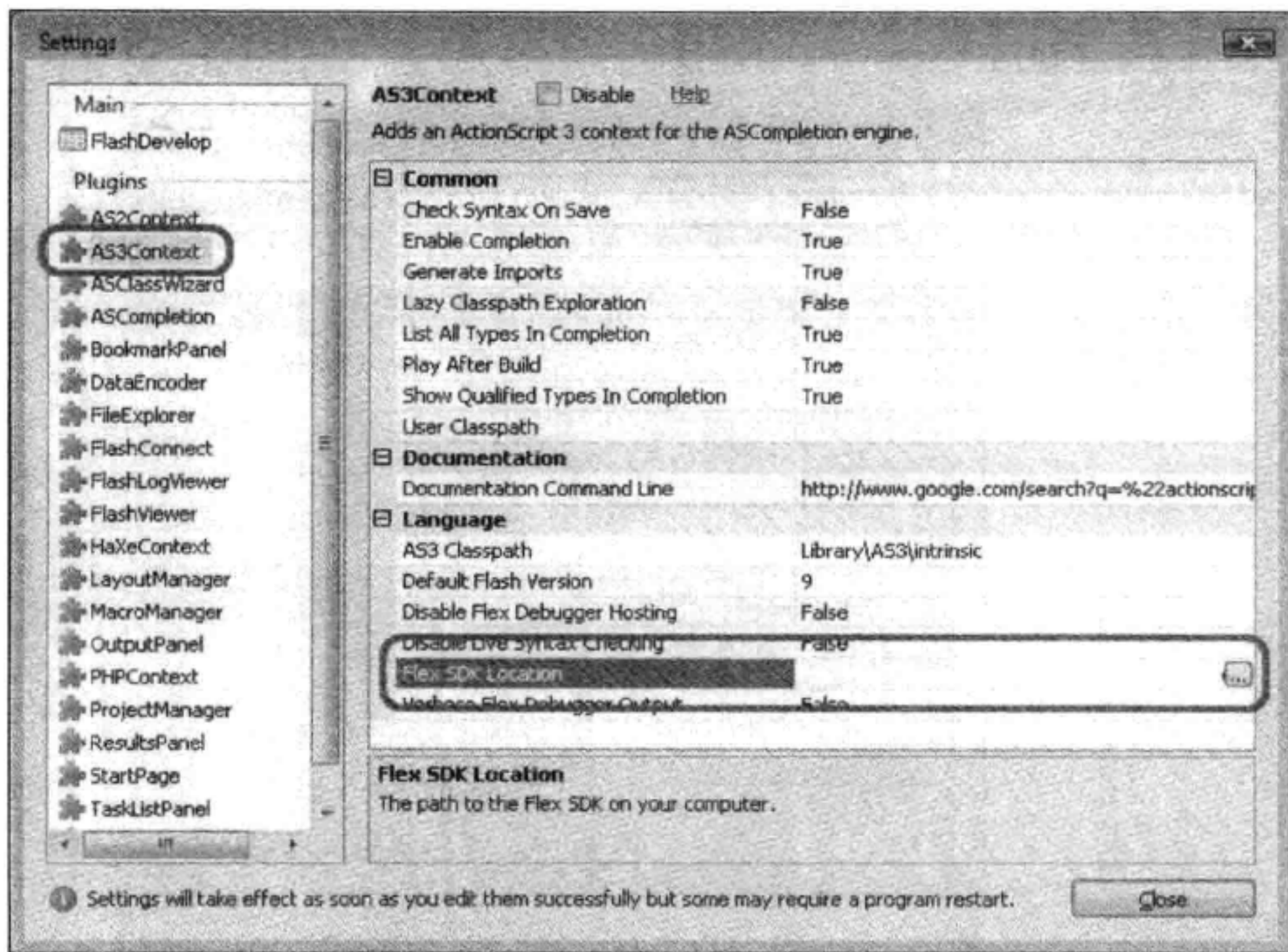


图 1-8 Settings 对话框

(3) 浏览到 Flex SDK 解包的文件夹,单击 OK 按钮。

(4) 现在 Flex SDK 解包的文件夹应显示在更新的位置,单击 Close 按钮。

(5) 打开 FlashDevelop Project,执行 Project|Properties 命令。

(6) 在 Output 选项卡中,单击 Target 下拉列表框,选择 Flash Player 10,单击 OK 按钮,保存设置。

1.6.3 Adobe Flash CS4

在 Adobe Flash CS4 中使用 Flex SDK 的步骤如下。

- (1) 单击 Edit|Preferences 命令。
- (2) 在左边的 Category 面板中,选择 ActionScript 选项,然后,单击 ActionScript 3.0 Settings 按钮。
- (3) 在如图 1-9 所示的 Flex SDK Path 文本框内,输入 Flex SDK 解包的文件夹位置,或者单击文本框旁的“文件夹”图标按钮,浏览到 Flex SDK 解包的文件夹位置。
- (4) 单击 OK 按钮,保存配置的改变。
- (5) 单击 OK 按钮,关闭 Preferences 窗口。

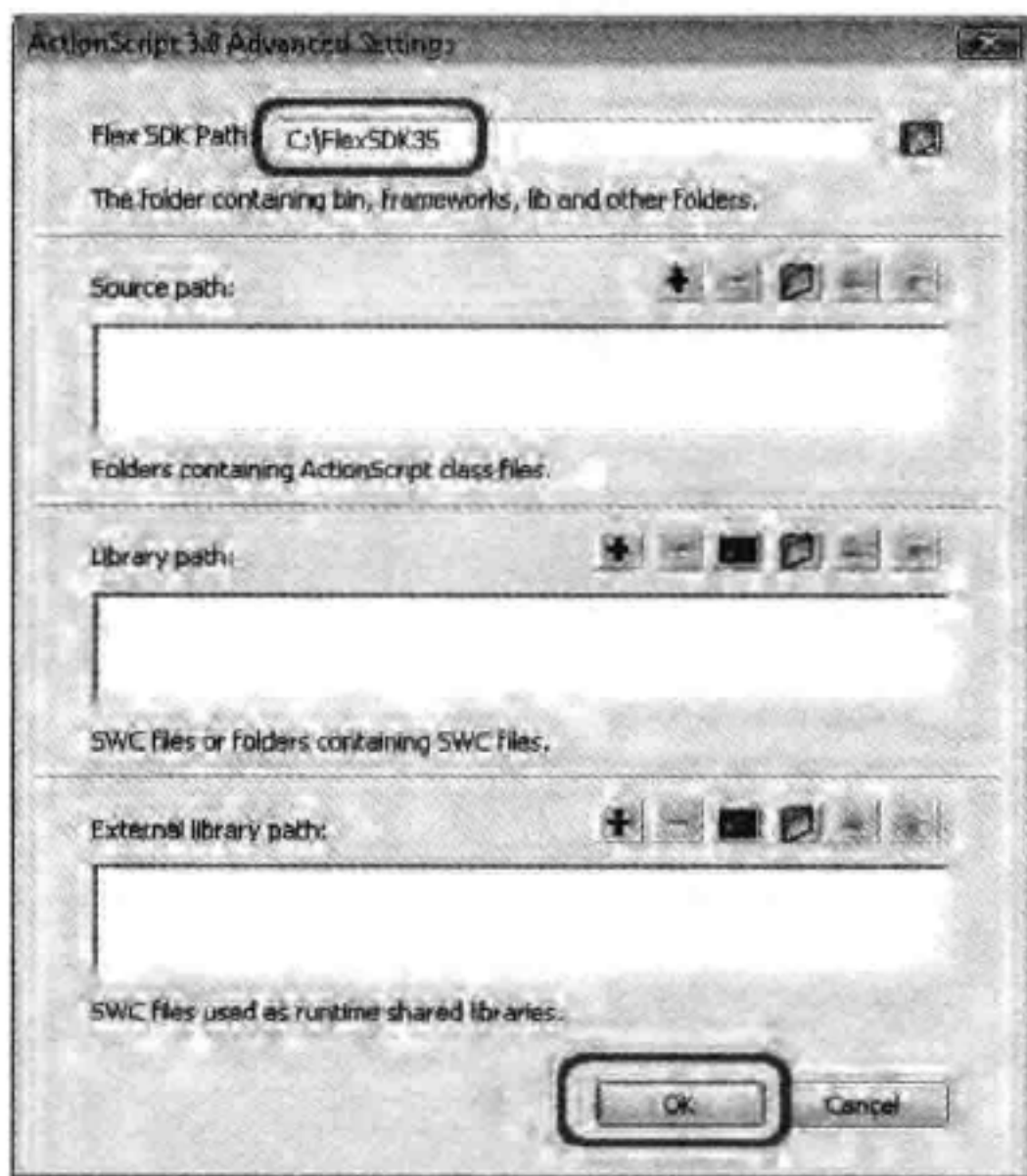


图 1-9 ActionScript 3.0 Advanced Settings 对话框

1.7 建立初始的应用程序

到现在为止,已经完成配置编写程序工具,新的工程项目已建立,配置了项目能够使用的 Away3D 库,以及针对能运行 Flash Player 10 的 Flex SDK 版本。现在是建立第一个 Away3D 应用程序的时候了。

Away3D 功能很强,但在使用这些功能之前,必须建立几个基本类。

(1) Scene3D,它代表保存显示在屏幕上的 3D 对象的三维空间,这通常被称为场景(Scene)。

(2) Camera3D,它提供了一个对象,通过 Camera3D 对象,场景能视察到,这通常被称为照相机(Camera)。

(3) View3D,它是一个容器,它添加到显示场景的舞台,通过照相机观看 View3D,能看到场景。这通常被称为视口(View)。

全书使用的 3D Object 这个术语,是个普通的术语,将它引用放到场景中的任何 3D 模型或原始形状。3D Object 由 Object3D 类表示。Object3D 类在 Away3D.core.base 包中,许多类继承了 Object3D 类。

Away3DTemplate 类包含在建立和初始化 Scene, Camera 和 View 这些对象的时候需要的基本逻辑。下面看看组成 Away3DTemplate 类的代码:

```
package
{
    /* 场景、照相机和视口这些对象,各自分别由 Scene3D 类、Camera3D 类和 View3D 类代表,在
    Away3DTemplate 类的包里首先要把它引入,以便它们是可用的 */
    import Away3D.cameras.Camera3D;
    import Away3D.containers.Scene3D;
    import Away3D.containers.View3D;
    /* 同时还要引入 Flash 的精灵类 Sprite,它们是 Away3DTemplate 类要继承的,还要引入 Flash
    事件类 Event,这些类是 Flash 事件响应系统要用的 */
    import flash.display.Sprite;
    import flash.events.Event;
    /* 通过继承 flash.display.Sprite, Away3DTemplate 类能添加到 Flash 的舞台,就像其他可视化
    元素一样 */
    Public class Away3DTemplate extends Sprite
    {
        /* 通过引入的这些类,为场景、照相机和视口对象定义属性 */
        protected var scene:Scene3D;
        protected var camera:Camera3D;
        protected var view:View3D;
        /* 构造函数继续调用一些其他函数,每个函数用于建立 3D 应用程序的某个方面的应用 */
        public function Away3DTemplate():void
        {
            /* 首先建立用户接口元素,由 initUI()函数来建立 */
            initUI();
            /* Away3D 引擎的核心元素的这些属性(即场景、照相机和视口),通过调用 initEngine()函数,
            进行初始化 */
            initEngine();
        }
    }
}
```

```

    /* 场景由 initScene()函数来构成 */
    initScene();
    /* 事件侦听器的配置,由初始化侦听函数 initListeners()来进行 */
    initListeners();
}
/* initEngine()函数,是建立 Away3D 引擎的核心元素的函数 */
public function initEngine():void
{
    /* 须明确建立的唯一对象是 View3D 类的对象 */
    view=new View3D();
    /* 场景、照相机对象由 View3D 类通过默认来建立,也可手工建立,这里为了方便直接引用这两个对象

```

通过默认建立的照相机位置,在 Z 轴负方向 1000 单位处(即它的位置是 (0,0,-1000),回头朝向场景中心。有关 3D 场景位置的更多信息,请参见 1.9 节。

```

    scene=view.scene;
    camera=view.camera;
    /* 就像自己创建的 Away3DTemplate 类一样,View3D 类也要继承 Flash 精灵类 Sprite,以便
    View3D 对象在屏幕上是可视的,必须把它添加成一子对象 */
    addChild(view);
    /* 最后,将 View3D 对象定位到屏幕中心,请注意:这里赋给 View3D 对象的 X 和 Y 坐标位置是在
    Flash 舞台中的,而不是相对于 3D 场景里的 */
    view.x=stage.stageWidth/2;
    view.y=stage.stageHeight/2;
}
/* initListeners()函数用于注册处理事件的句柄 */
public function initListeners():void
{
    /* 一旦在每个帧里发生 Event.ENTER_FRAME 事件,就要调用响应事件的注册函数
    onEnterFrame() */
    addEventListener(Event.ENTER_FRAME,onEnterFrame);
}

```

在 onEnterFrame()函数里,通过调用 View3D render()函数,在屏幕上画出一个画面,请记住,onEnterFrame()函数是在响应 Event.ENTER_FRAME 事件时连续被调用的,以这样一种方式连续地画出很多画面,就能在 3D 场景里,建立运动的动画效果,就像在屏幕上投影的电影画面一样。

```

public function onEnterFrame(event, Event):void
{

```



```
view.render();  
}
```

initScene()函数是用来构成场景的,在运行应用程序的时候,它让人们能查看场景里的一些 3D 动画。但因为人们希望把 Away3DTemplate 类设计成一般化的类,这个 initScene()函数是空的。预期以后创建一个继承 Away3DTemplate 的类,实现其 initScene()函数功能。

```
protected function initScene():void  
{  
}  
/* initUI()函数是建立任何用户接口的元素,与 initScene()函数一样,initUI()也是空的,预期以后  
创建一个继承 Away3DTemplate 的类,实现 initUI()用户接口的功能 */  
protected function initUI():void  
{  
}  
}  
}
```

在一般的情况下,通过引用 initEngine()函数来建立并初始化场景、照相机和视口三个函数,它不会管要建立的应用是什么类型的,是游戏,是 3D 用户接口,还是一个广告或标语。在 Away3D.test 包里,有一叫做 SimpleView 的类,通过初始化这三个类,它允许很快地建立并运行程序。不但要把 Away3DTemplate 设计得易于使用,还要与 SimpleView 类有相同的最终结果,作为贯穿本书的其他全部的将要创建的应用程序的基础类。

1.8 运行 Away3DTemplate

本节通过以下过程建立空的工程项,以容纳使用 Away3D 引擎编写代码。为运行 Away3DTemplate 类,必须将它添加到空的工程项中。还必须指明 Away3DTemplate 类为应用程序的入口,以便当应用程序开始运行时它能执行。

1.8.1 Adobe Flex Builder 和 Adobe Flash Builder

把 Away3Dtemplate 类添加到前面建立的空的工程项内的步骤如下。

(1) 当用原先的步骤建立空的工程项的时候, Flex /Falsh Builder 会建立一个默认的与

工程项同名的 ActionScript 文件,例如 Away3DTemplate.as,这个文件必须要删除,为此要右击这个文件,在 Project Explorer|Flex Navigator 内,单击 Delete 选项。

(2) 单击 Yes 按钮,删除文件。

(3) 执行 File|New|ActionScript Class 命令。

(4) 在 New ActionScript Class 对话框里的 Name 文本框内输入 Away3DTemplate,单击 Finish 按钮。

(5) 在新建 ActionScript 项目的 src 目录下,要引入下载解包后的 D:\Away3D_3_6_0\Away3D_3_6_0\src\Away3D 文件夹,以便能导入 Away3D.* 的包。

(6) 把上面写的 Away3DTemplate 类的代码,复制粘贴到新的 Away3DTemplate.as 文件中,覆盖默认产生的代码。

(7) 右击 Project Explore|Flex Navigator 内的 Away3DTemplate.as 文件,单击 Set as Default Application 选项,此时这个文件的图标应包含很小的绿色的三角和蓝色的圆球,如图 1-10 所示。



图 1-10 Away3DTemplate.as 文件图标

(8) 双击 Away3DTemplate.as 文件,开始编译并运行应用文件。

1.8.2 FlashDevelop

把 Away3DTemplate 类添加到已建立的项目中的步骤如下。

(1) 用前面讲述的方法,建立空的项目,FlashDevelop 将在 src 目录中建立默认的名字为 Main.as 的文件,该文件应删除。右击 Project 面板内的 Main.as,选择 Delete 选项。

(2) 单击 OK 按钮,确认删除该文件。

(3) 执行 File|New|AS3 Document 命令。

(4) 把 Away3DTemplate 类的代码粘贴到这个新文件中,并覆盖原默认生成的代码。

(5) 选择 File|Save|Save as 命令,将文件命名为 Away3DTemplate.as 的项目保存在 src 目录下,如图 1-11 所示。

(6) 右击在项目面板下的 Away3DTemplate.as 文件,单击 Always Compile 选项,该文件的图标应改变成包含一个绿色向下的箭头。

(7) 执行 Project|Test Movie 命令,编译,运行此程序。



图 1-11 Project 面板

1.8.3 Adobe Flash CS4

把 Away3DTemplate 类添加到已建立的 Adobe Flash CS4 项目中的步骤如下。

- (1) 运行 Adobe Flash CS4, 并选择 File|New 选项。
- (2) 选中 General 选项卡中的 Flash(ActionScript 3.0)文件, 并单击 OK 按钮。
- (3) 将 Away3DTemplate 类的代码粘贴到新文件中。
- (4) 选择 File|Save 选项。
- (5) 把文件 Away3DTemplate.as 保存到原先建立的同名 FLA 文件的目录中。
- (6) 返回到文件保存的目录中, 选中 Away3DTemplate.FLA 文件, 再执行 File|Publish Settings 命令。
- (7) 单击 Flash 标签, 单击 Settings 按钮。
- (8) 在如图 1-12 所示的对话框内的 Document class 文本编辑框中输入 Away3DTemplate, 单击 OK 按钮保存改变并关闭 Advanced ActionScript 3.0 Settings 对话框。

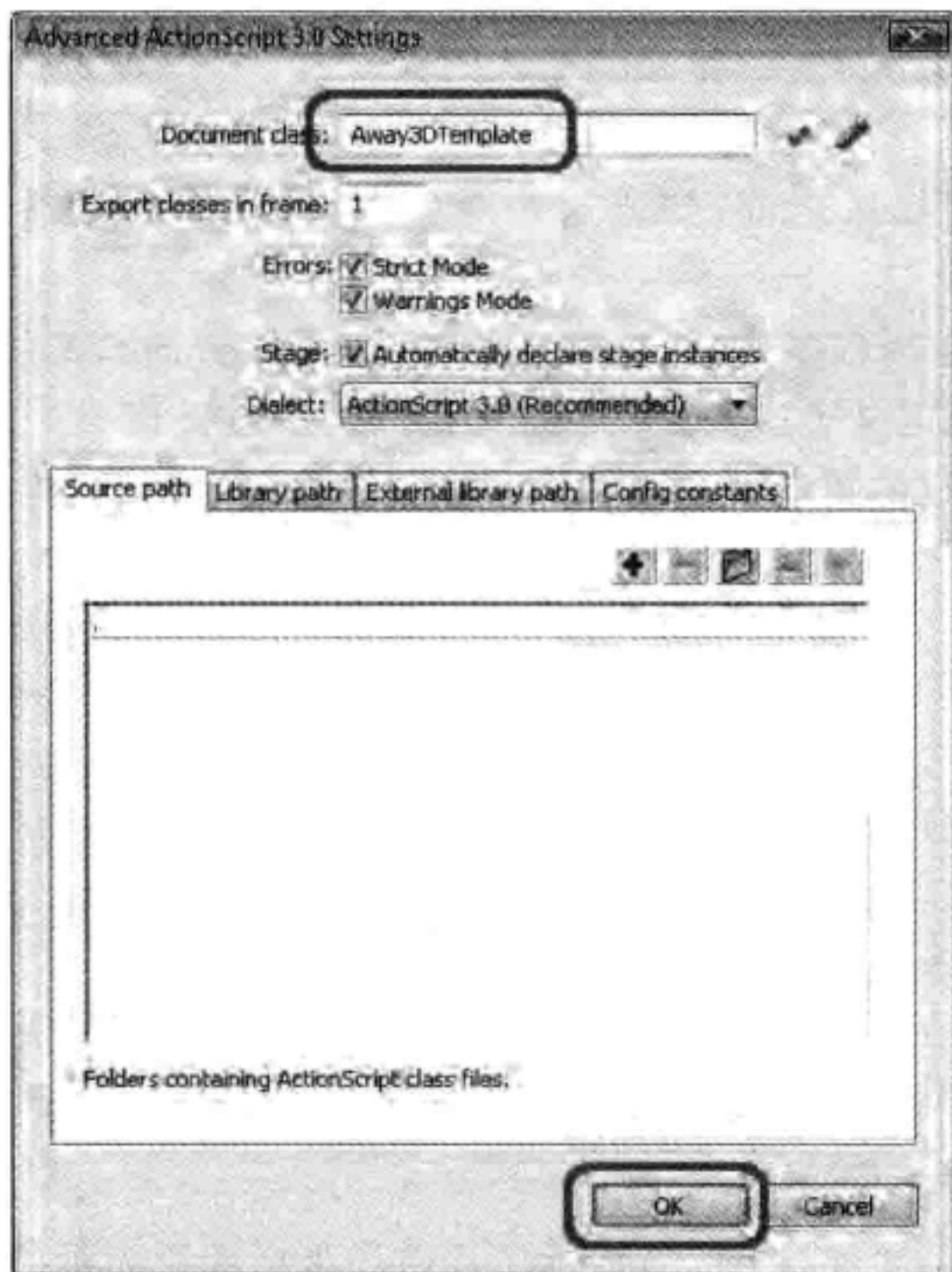


图 1-12 高级设置

(9) 单击 OK 按钮,关闭 Publish Settings 窗口。

(10) 为了编译并运行应用程序,执行 Control | Test Movie 命令,生成 Away3DTemplate.swf 文件和 Away3DTemplate.html 浏览器文件,并弹出效果窗口。

在第(8)步中,当单击 OK 按钮时,如果看到一个 ActionScript 的警告框,要返回到第(5)步,确保保存的 Away3DTemplate.as 文件与 FLA 文件在相同的目录中。

1.8.4 最终结果

当编译并运行这个应用程序的时候,会看到什么也没有,然而,这是对的,这正是我们所希望的结果。Away3DTemplate 类只是提供了一个基础,它注重 Away3D 引擎的初始化和更新,但没有建立任何在屏幕上看得到的可视化的对象。

1.9 在一个 3D 场景中定位对象

在开始添加对象到场景之前,知道这些对象在一个 3D 环境中是如何定位的是很重要的。

传统的 2D Flash 应用程序,把这些对象沿 X 轴(水平方向),Y 轴(垂直方向)放到屏幕上。笛卡儿坐标唯一地定义了这些对象在二维空间里的位置。

Away3D 扩展了 2D 的笛卡儿坐标系统,添加了第三个坐标轴 Z 轴,它允许对象定义景深。

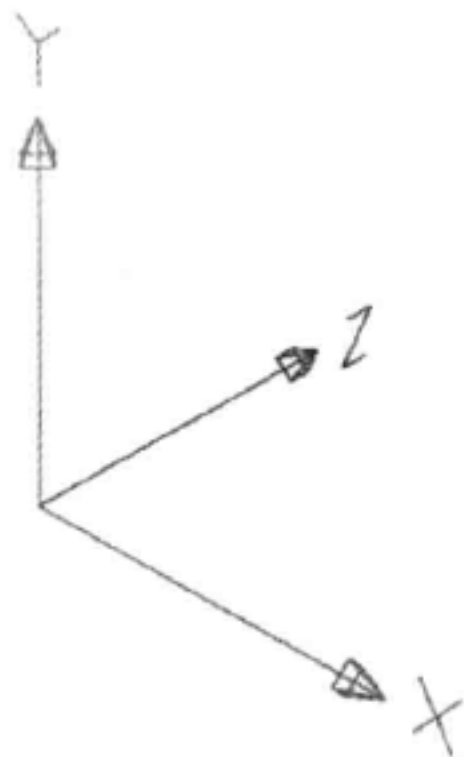


图 1-13 左手坐标系统

三维坐标系统,通常被引用为左手系统或右手系统,Away3D 使用左手坐标系统。为形象化理解左手坐标系统,可伸出左手,使掌心背向自己,中指指向背向自己与掌心的方向相同,使无名指直立向上指向空中,使大拇指指向右方。此时,中指指定的方向是 Z 轴,无名指指定的方向是 Y 轴,而大拇指指定的方向是 X 轴。各个指头朝向坐标轴的正端方向。

当在空中如此用手指指向时,也许看起来是一个有趣的练习,但它实际上是一种容易展示使人明白三维(简称 3D)空间的工作方式。

图 1-13 演示了用左手坐标系统表示的三个坐标轴,注意:坐标系统的 Y 轴是垂直的。与传统的 Flash 二维坐标系统比较,在

Flash 把一个对象放到屏幕较低的地方,就意味给它指定一个较高的 Y 轴的位置;但在 Away3D 里,对象具有较高的 Y 值,表示它将沿 Y 轴升高向上。

1.10 继承 Away3DTemplate 类构建一个场景

为了实际上在屏幕上显示一个 3D 对象,必须建立另一个叫做 SphereDemo 的类,下面是 SphereDemo 类的代码。

```
package
{
    /* Away3D 包含很多的原始型体,它们很容易地添加到场景里.涵盖这些标准基本体的详尽细节,
    请参见第 2 章.我们将把原始型体球添加到场景里,它们由 Sphere 类表示 */
    import Away3D.primitives.Sphere;
    /* SphereDemo 类继承 Away3DTemplate 类,允许用最少的代码初始化并更新 Away3D 引擎 */
    public class SphereDemo extends Away3DTemplate
    {
        /* 在 SphereDemo 构造函数中,用 super()语句直接调用 Away3DTemplate 的构造函数,它将转而
        初始化 Away3D 引擎,这要调用 initUI()、initEngine()、initScene()和 initListeners()函数 */
        public function SphereDemo()
        {
            super();
        }
        /* initScene()函数,以前在 Away3DTemplate 类里,故意地留下一空场景.在 SphereDemo 类里,把
        sphere 添加到场景,覆盖 initScene()函数 */
        protected override function initScene():void
        {
            /* 首先,从基类调用 initScene()函数,
```

实际上,从基类调用 `initScene()` 函数,什么事也不会做,因为以前在 `Away3DTemplate` 类里,故意地留下一空场景。无论如何,调用此基类函数是个好习惯,因为其他的函数需要它,如 `initEngine()`和 `initListeners()`

```
super.initScene();
/* 当建立一个新的 Sphere 三维对象的时候,默认将该对象放到场景原点.如果还记得的话,
Away3DTemplate 类把照相机 Camera 开始时是放在(0,0,-1000)的坐标位置,并朝向场景的原点,
这就意味着,当运行应用程序时,Sphere 三维对象在照相机 Camera 的前面
```

很多 `Away3D` 类作为一个构造函数的参数,被调用为一个初始化的对象,而建立此对象的实例。使用对象的文字符号法,建立这些初始化对象。下述的代码建立 `Sphere` 对象,并把它放在(0,0,500)的位置 */

```

var sphere:Sphere=new Sphere(
    {
        x:0;
        y:0;
        z:500;
    }
)
/* 对象的文字符号是建立关联数组的缩记方法,关联数组是各个属性映射到各个值的对象实例.以下的代码与上述的代码有相同的效果 */
var obj:Object=new Object();
obj.x=0;
obj.y=0;
obj.z=500;
sphere=new Object(obj);
/* Sphere 对象在实例化之后,它的各个属性值也要设置 */
var sphere:Sphere=new Sphere();
sphere.x=0;
sphere.y=0;
sphere.z=500;
/* 在这里,虽然没有使用初始化对象 init Object,但它们在本书的其他地方被广泛地使用 */
var sphere:Sphere=new Sphere();
/* 最后,为了能看见 sphere,必须把它添加到子场景中 */
scene.addChild(sphere);
}
}
}

```

1.10.1 运行 SphereDemo 应用程序

对于所有用程序编写工具编写的在 Away3DTemplate 类基础之上而创建的应用程序,都要涉及建立一个新的 ActionScript 文件,把新应用类写到这个文件里去,并当程序执行时,把这个新类当作运行的开始点。

运行 SphereDemo 的过程,可以用于运行本书后面介绍的任何其他例子。这个过程非常类似于原来建立 Away3DTemplate 的过程,只是名字改变了而已。

1. Adobe Flex 和 Adobe Flash Builder

遵循 Adobe Flex 下管理运行 Away3DTemplate 的第(3)~(7)步指定的命令,只要在相应的地方,把 Away3DTemplate 的名字替换成 SphereDemo,这将导致在项目中,既有 Away3DTemplate 类和 SphereDemo 类,也有它们的 AS 源文件,并且把 SphereDemo 类设置成默认的应用程序的入口。

2. FlashDevelop

遵循 FlashDevelop 下管理运行 Away3DTemplate 的第(3)~(7)步指定的命令,只要在相应的地方,把 Away3DTemplate 的名字替换成 SphereDemo,这将导致在项目中,既有 Away3DTemplate 和 SphereDemo 类,也有它们的 AS 文件,并且 SphereDemo 类设置成总是被编译的类。

3. Adobe Flash CS4

遵循 Adobe Flash CS4 下管理运行 Away3DTemplate 的第(1)~(10)步指定的命令,只要在相应的地方,把 Away3DTemplate 的名字替换成 SphereDemo,这将导入两个文件: SphereDemo.as 和 Away3DTemplate.as,保存在与 FLA 文件相同的目录下,并且 SphereDemo 类的使用成为应用程序的入口点。

1.10.2 最终结果

现在编译并运行应用程序的时候,就会看到一个圆球出现在屏幕中央,通过继承 Away3DTemplate 类,仅用很少的几行代码,就建立了 SphereDemo 的三维样本例子。这个例子证明,可以用 Away3DTemplate 类作为基类,快速建立一个新的 Away3D 应用程序。

本书以后介绍的全部例子,都将在 Away3DTemplate 类的基础上构建,与这里构建 SphereDemo 类的方式基本一样。

建立并显示原始3D模型

在第1章中介绍了如何建立一简单的 Away3D 应用程序,并使之运行,办法就是建立一个球,并把它添加到场景里。这个球正是 3D 基类模型中的一个。包含在 Away3D 中的原始 3D 模型,还有一些通用的模型,如立方体、平面体和圆锥体以及一些非通用的图形,如海龟等。在这一章中读者会看到,有很多可用来建立原始 3D 模型的类,通过建立简单的应用,把原始 3D 模型添加到场景中,即可创建响应键盘输入的应用程序。

在深入到这些原始 3D 模型之前,将首先学习构建 3D 对象的基本元素:顶点、三角表面、Sprite3D 对象和线段。这些基本元素用于构建更复杂的 3D 模型。此外,还将浏览 UV 纹理坐标系统,它定义了如何把纹理映射应用到 3D 对象的表面。

本章主要内容:

- 组成一个 3D 对象的基本元素
- UV 坐标系统
- 建立和显示包含在 Away3D 中的原始 3D 模型

2.1 一个 3D 对象的基本元素

通过 Away3D 显示的每一个 3D 对象,实际上是由很多基本元素的集合组成的,它们联合构成了人们在屏幕上看到的图形。以下 4 个基本元素用于构造更复杂的 3D 对象。

- (1) 顶点 Vertices。
- (2) 三角面 Triangle faces。

(3) Sprite3D 对象。

(4) 线段 Segments。

这些元素中有一些是不能直接看见的,但它们扮演着建立最终模型的至关重要的角色。下面看看这些基本元素,以及这些基本元素是如何在一起构建一个 3D 对象的。

2.1.1 顶点

顶点是 3D 空间里的一个点,它由沿 X、Y 和 Z 坐标轴的坐标值定义。一个单个的顶点,并没有体积和形状,且在场景中是不可见的,但是它们能联合用于构成 3D 图形的各个拐角。顶点由 away3d.core.base 包中的 Vertex 类表示。

2.1.2 三角面

三个点(如图 2-1 所示)就可以用于定义一个三角形(如图 2-2 所示),即众所周知的三角面。



图 2-1 顶点



图 2-2 三角形

三角面是 Away3D 能够直接添加到网格构建复杂图形的三元素之一。可以将一个网格想象为一个容器,用于保持这些基本元素的一个集合,允许它们显示一组可视的、更加可辨认的 3D 对象。网格由 Meshes 类表示,它们包含在 Away3D.core.base 包中。

用很少数量的三角面,能组成一个简单的网格。如图 2-3 所示,图中的立方体网格由 12 个三角面组成,它的每一个面是两个三角面。

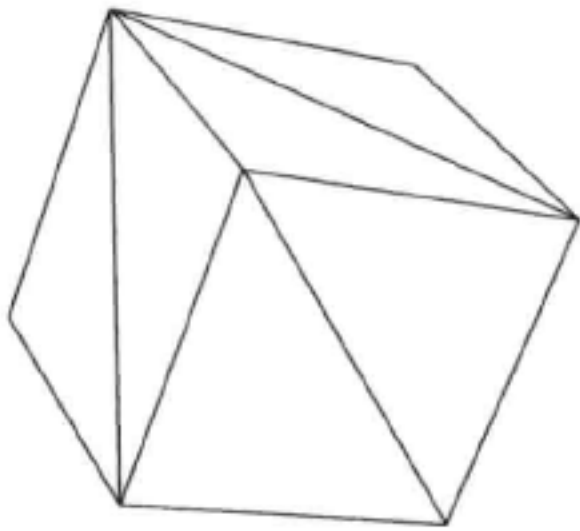


图 2-3 立方体网格

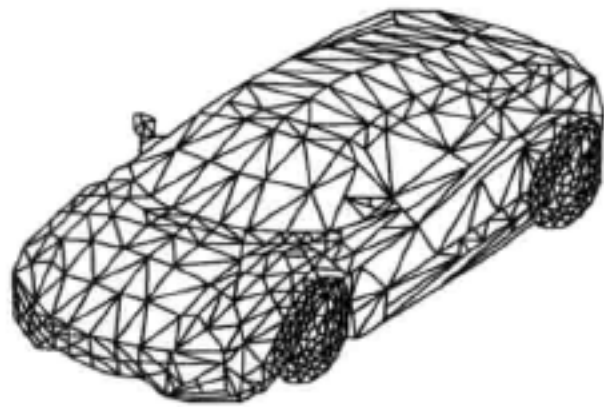


图 2-4 小汽车的三角面结构

对于更复杂的一些网格,例如,如图 2-4 所示的小汽车可以由几千个三角面组成。

一般来讲,越是复杂的 3D 对象, Away3D 处理它,所花费的时间越长。于是在现实世界的应用里,应该将可视的复杂网格限制在一定的数量范围内,以便于维持一个使用者可接受的运行时间。

三角面由包含在 away3d. core. base 包中的 Face 类表示,这里是一些编码例子,用于建立一个仅容纳一个三角面的网格。

首先,必须建立一个新的实例 Mesh. class 类,它将保存三角面元素。

```
var mesh:Mesh=new Mesh();
```

下一步,建立一个新的 Face. class 类实例,它代表添加到网格中的三角面元素。

```
var face:Face=new Face(
/* 首先要有三个参数,传递到 Face 构造函数里,定义三个顶点,组成三角形 */
new Vertex(0,10,0),
new Vertex(-10,-10,0),
new Vertex(10,-10,0),
/* 第 4 个参数,用于定义使用到这个元素的材质,更详细的细节,请参见第 4 章,这里仅传递 null,表示三角面将使用默认的网格材质
```

默认的施加于网格的材质,是一个金属线色材质 WireColorMaterial 类的实例,它显示的颜色是每次在金属线色材质类实例化时,其参数用随机函数指定的。这表示,在这里建立的三角面,在每次应用程序运行时会显示不同的颜色。

```
null;
/* 最后的三个参数,定义了使用到三角面的 UV 纹理贴图坐标系统,它定义了纹理材质是如何使用到三角面上的,后面介绍 UV 纹理贴图坐标系统时,会详细讨论 */
new UV(0,0),
new UV(0,1),
new UV(1,0)
};
```

最后把三角面添加到网格里去,作为它的一部分。

```
mesh.addFace(face);
```

这使网格 Mesh 对象转而添加到场景里。

在三角面里面定义顶点的顺序,决定了三角面的哪一面是前面,哪一面是背面。这个区分是非常重要的,因为默认的情况下,它的背面是看不到的。

用反时针的方向顺序排列顶点的这一面,是三角面的前面。图 2-5 显示了 Vertex 对象第一次传递给 Face 构造函数的三个顶点参数,并且是反时针方向的,这定义了三角面的前面和背面。

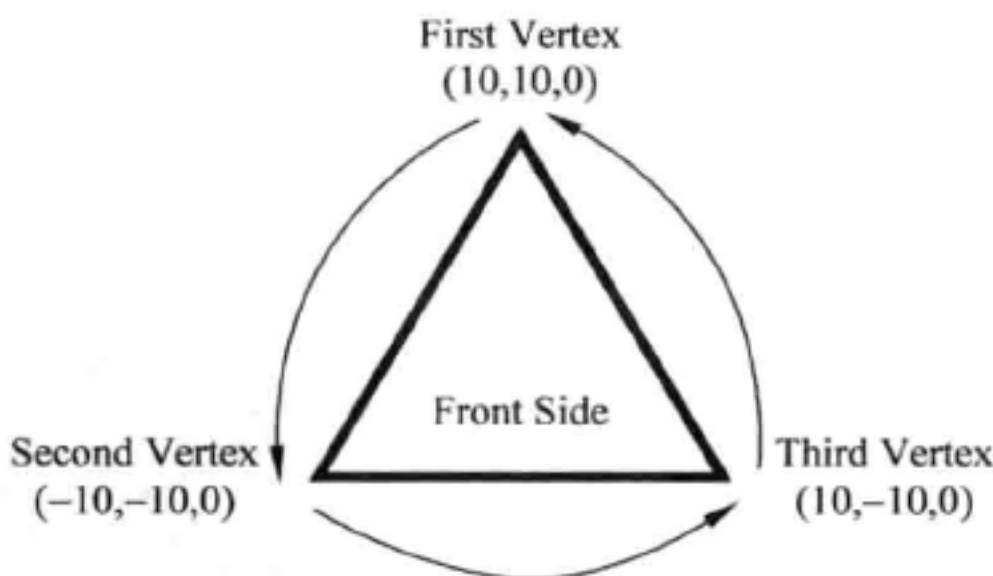


图 2-5 顶点对象传递给面构造函数的参数

建立三角面,并把它添加到网格里,是最基本的过程,告知如何结构化大多数包含在 Away3D 里的原始 3D 模型。虽然有一些原始 3D 模型类,例如立方体 Cube 类,继承 away3d.primitive 包的 AbstractPrimitive 类时,利用了大量的 AbstractPrimitive 类提供的工具函数,用于辅助建立原始 3D 模型立方体。

2.1.3 Sprite3D 精灵

很多网格也能容纳 Sprite3D 对象,这些对象是二维的矩形,它们总是朝向照相机,然而 Sprite3D 对象是没有深度的,因为绝不可能从另一面看到它,由于缺乏深度,它绝不可被照相机察觉到。

Sprite3D 类存在于 away3d.sprites 包里。以下是个例子,它建立一个 Mesh 网格对象,包含一个位于网格对象原点的 Sprite3D 对象。

```
var mesh:Mesh=new Mesh();
```

Sprite3D 对象构造函数的第一个参数是它显示时使用的材质,这是个可选的参数,默认值为 null,没有材质的 Sprite3D 对象,仅显示为一个矩形。为使这个例子更有趣,下面提供一个新的 BitmapMaterial 类实例,它设置显示一嵌入式的 MyEmbeddedTexture 的图像文件。材质、Cast 类和嵌入资源,详见第 5 章。

```
var sprite:Sprite3D=new Sprite3D() {  
    new BitmapMaterial(Cast,bitmap(MyEmbeddedTexture));  
};
```

然后,Sprite3D 对象,用 addSprite()函数添加到它的父网格中。

```
mesh.addSprite(sprite);
```

Sprite3D 对象能用来表现来自任意方向的同样细节,如雪花或圣诞树上的装饰球。Sprite3D 对象是一个非常简单的元素,因此能很快地画出来。使用 Sprite3D 对象代替一组三角面,执行效率会有很大的提高。

在图 2-6 和图 2-7 中,采取网格表示地球,这也许是天文学里设计表现太阳系的应用程序的一部分,典型的是用原始 3D 模型中的球建立的。如图 2-6 所示的线框图像,表现的是用三角面组成的球体,图 2-7 表现的是带有用到它上面的地球纹理的球体。

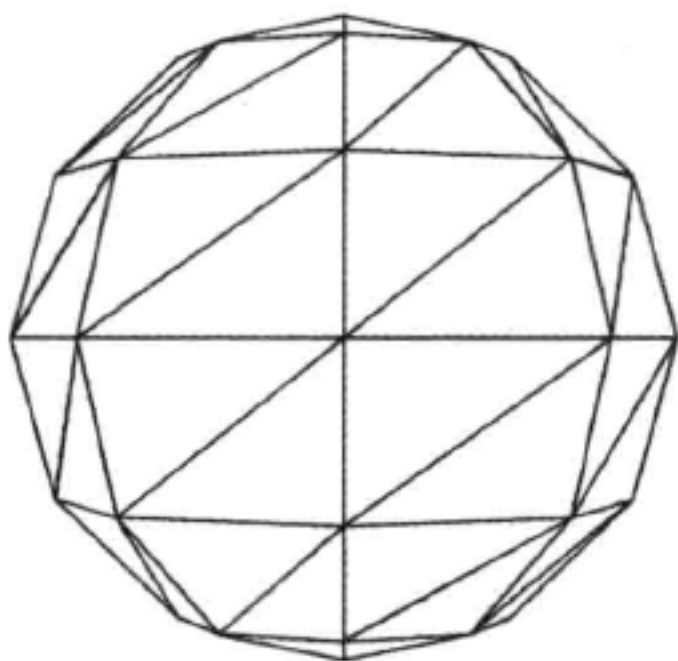


图 2-6 球体的线框表示



图 2-7 带纹理的球体

因为这个球体是由少量的三角面建立的,这就表示它在显示时会有很多尖锐的有角的边缘。因为三角数量少,可能希望 Away3D 应用程序在显示一个球时,运行性能好一些。但是,如果显示全部行星,那将是怎样呢?你也许希望显示全部行星以及围绕它的全部卫星,画出的三角面等于组成全部球的三角面总和相加起来。

现在看一看图 2-8,是包含一个 Sprite3D 对象的网格。



图 2-8 包含 Sprite3D 对象的地球网格

Sprite3D 对象显示一个表示地球从单一视角看到的材质,使用此纹理的材质是预先由几千个三角面组成的一个球描绘出来的,它消除了原先在三角网格里看到的有角的边缘。Away3D 类能够画出 Sprite3D 对象,显示预先描绘出来的纹理,比上面第一次表示低质量的那两个图形球的显示速度要快得多。用于显示太阳系中的行星及围绕它们的卫星,如果显示它们用的是单个 Sprite3D 对象,组成 Sprite3D 对象的这些网格,与使用的三角面的这些网格相比,使用 Sprite3D 对象的程序具有更快的运行性能。

Sprite3D 对象的缺点是与正规的 3D 对象不同,当从不同的角度来观察时,它显现的都相同,即使从另一边来观察用 Sprite3D 对象组成的地球的网格,它仍然将显示印度洋!

Away3D 包含一个叫做 DirectionalSprite 的类,它允许平面对象,以变化的不同的观察视角,而显示不同的图像。更详细参见第 9 章。

2.1.4 段

段用于显示 2D 直线。严格地讲,一条二维直线是没有体积的,因此应该是不可看见的。然而在很多情形下,在场景中能看到一条直线是非常有用的。

段由 Away3D.core.base 包内的 Segment 类表示,段的几何意义是由两顶点定义的。下面是一个例子,建立一个网格对象 mesh,它包含一个段对象,段对象沿网格原点 X 轴方向两边各延展 50 个单位宽度。

```
var mesh:Mesh=new Mesh();
var segment:Segment=new Segment()
{
    new Vertex(-50,0,0),
    new Vertex(50,0,0)
};
```

利用 addSegment() 函数将段添加到它的父网格中:

```
mesh.addSegment(segment);
```

2.2 UV 纹理贴图坐标系统

使用 Bitmap 材质显示纹理映射时,通常是把 JPG、PNG 和 GIF 的图像文件映射到 3D 对象的表面上,三角面上纹理映射的位置取决于 UV 坐标系统的使用。这个名字来自于组成坐标空间的坐标轴,这些 2D 二维坐标有 0~1 的范围,用于把一个三角形的顶点映射到

与之相对应的纹理图上的映射位置。

图 2-9 展示了三个点的 UV 纹理坐标,这三个点就是前面建立三角面的三个顶点,映射到国际象棋棋盘纹理图上的样子。

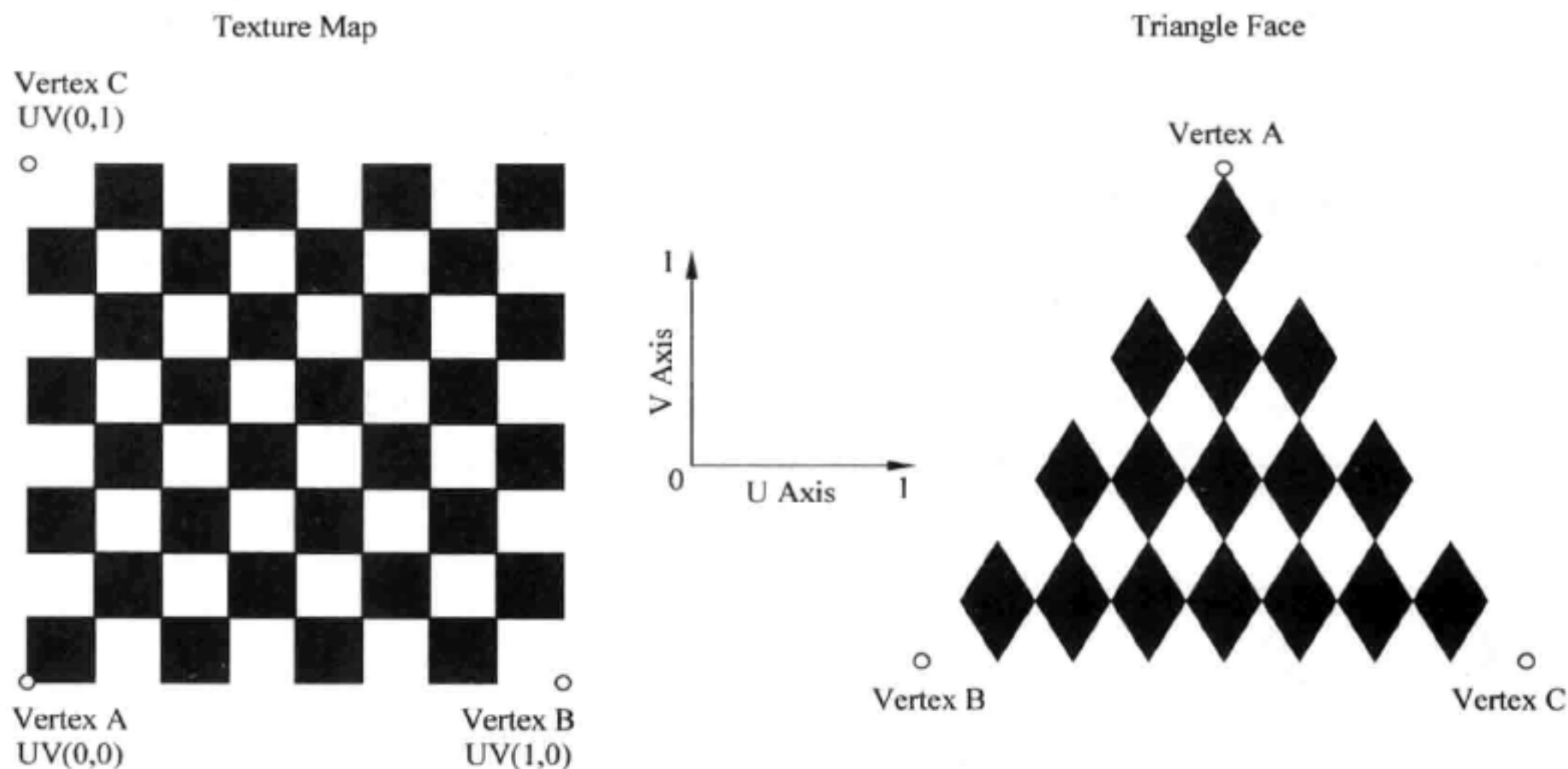


图 2-9 顶点的 UV 纹理坐标表示

U 和 V 坐标轴与标准的 Flash 用的 X 轴和 Y 轴的工作方式相同,坐标轴的命名很明确地区分了标准坐标系统与 UV 坐标系统间的区别,纹理图上那些标绘点映射到 XY 坐标,标绘在空间或 Flash 舞台上的点,请务必牢记:当 V 轴向上移动时,而在 Flash 舞台中的 Y 轴就与此相反,向下移动。反之,当 Flash 绘画向上增加时,则 V 轴向下移动。

2.3 创建原始的 3D 对象

我们已经看到一个复杂的模型,能够由一些顶点、三角面、Sprite3D 对象和段组成,在实践中,每一个模型都必须手工建立,而且每次还要使用构建这些模型的基本元素,这样将是冗长乏味且易于出错的工作任务,幸运的是 Away3D 包含大量的类,这些类能够用于建立广泛的、容易使用的基类模型。

为了展示包含在 Away3D 中的这些基类模型,下面建立一个应用程序,可在屏幕上有选择地显示它们。这是通过调用 PrimitivesDemo 类实现的。PrimitivesDemo 类继承了在第 1 章中介绍的搭建第一个 Away3D 应用程序中的 Away3DTemplate 类。

PrimitivesDemo 类相当大,但是它的函数很简单。下面分解它的代码,看看它是如何工作的。

```
package
{
/* Object3D 类必须引入, Object3D 类是所有加到场景类中的 3D 类的基类 */
import away3d.core.base.Object3d;
/* 还有很多类要从 away3d.primitives 包引入, 这些类中每一个类代表一个基类模型, 它们包含在
Away3D Library 库中 */
import away3d.primitives.Cone;
import away3d.primitives.Cube;
import away3d.primitives.Cylinder;
import away3d.primitives.GeodesicSpere;
import away3d.primitives.GridPlane;
import away3d.primitives.LineSegment;
import away3d.primitives.Plane;
import away3d.primitives.RegularPolygon;
import away3d.primitives.RoundedCube;
import away3d.primitives.seaTurtle;
import away3d.primitives.Skybox;
import away3d.primitives.Skybox6;
import away3d.primitives.Sphere;
import away3d.primitives.Torus;
import away3d.primitives.Triangle;
import away3d.primitives.Trident;
/* 应用程序将要响应两个 Flash 事件. 第一个事件是每帧都要触发一次, 这帧就是在第 1 章搭建第
一个 Away3D 应用程序时, 用于重画并能动画地绘制场景的事件. 第二个事件是当键盘上的键释放
时, 它将用于改变原来在屏幕上的显示, 响应这些键盘上的键释放时的事件请求, 这些事件就是要引
入的事件类和键盘事件类, 这些类包含在 flash.events 包中 */
import flash.events.Event;
import flash.events.KeyboardEvent;
/* 某些基类模型有唯一的应用材质的方式. 为了展示这点, 我们将使用 BitmapFileMaterial 类, 有关
材质的内容详见第 5 章 */
import away3d.materials.BitmapFileMaterial;
/* 正像第 1 章中的 SphereDemo 类一样, 这里的 PrimitivesDemo 类也是继承 Away3DTemplate 类,
这使它很容易初始化 Away3D 引擎 */
public class PrimitivesDemo extends Away3DTemplate;
/* PrimitivesDemo 类有一个属性, 叫做 currientPrimitive, 它将引用一个 Object3D 类. 由于所有的
原始类都是直接或间接继承于 Object3D 类, 因此可以用这个属性去引用任何一个建立的原始 3D 对
象 */
protected var currientPrimitive:Object3D;
/* PrimitivesDemo 的构造函数什么也没做, 它不会比 Away3DTemplate 调用构造函数时做得更
多, 构造函数将转而调用 initUI()、initEngine()、initScene()和 initListeners()函数 */
public function PrimitivesDemo():void
{
    Super();
}
```



```

    }
    /* initEngine()函数重载了 camera 初始的位置, camera 初始的位置是在 Z 轴负方向 500 个单位, 这
    将给 camera 观察原始 3D 对象一个很好的位置, 原始 3D 对象被放在原点 */
    protected override function initEngine():void
    {
        super.initEngine();
        camera.z=-500;
    }
    /* initScene()函数重载了 initSphere()函数, 它将导致第一次显示时, 把原始球画到屏幕上 */
    protected override function initScene():void
    {
        super.initScene();
        initSphere();
    }
    /* 侦听器函数 initListeners()重载注册了被调用的 onKeyUp()函数, 此函数在响应键盘键释放阶段
    产生 KeyboardEvent.KEY_UP 事件. */
    protected override function initListeners():void
    {
        super.initListeners();
        Stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
    }
    /* 每个帧都要调用 onEnterFrame()函数一次, 这是动画发生的地方 */
    protected override function onEnterFrame(event:Event):void
    {
        super.onEnterFrame(event);
    }

```

为了从一个好的视角范围观察基类模型, 我们将在场景里慢慢地转动它们, 为此, 要修改 Object 类中的 rotationX、rotationY 和 rotationZ 属性, 有关以这样的方式绘制三维动画的更详细的内容, 请参见第 3 章。

在这个例子中, 3D 对象的旋转速度, 取决于应用程序帧的速率, 每秒画出的帧越多, 则旋转属性增加的越快, 因此 3D 对象旋转将会越快。

本章介绍如何使用 TweenLite 库以及如何实现 3D 对象的移动和旋转而与帧速无关的方式; 而在第 13 章中, 将介绍如何修改最大的帧速率。

```

        currientPrimitive.rotationX +=1;
        currientPrimitive.rotationY +=1;
        currientPrimitive.rotationZ +=1;
    }
    /* 舞台上, 当 KeyboardEvent.KEY_UP 事件发生时, 就要调用 onKeyUp()函数, 响应键盘上一个按
    键的释放 */

```

```
protected function onKeyUp(event:KeyboardEvent):void
{
/* 在一个新的基类模型建立并显示之前,原来当前正在显示的模型必须要从场景里移出.要完成这个任务,可以调用 removeCurrentPrimitive()函数 */
removeCurrentPrimitive();
/* 现在,响应刚刚释放的那个特定的键,键盘上的每个键,都由一个数字编码来标识.这些编码能够在网页: http://www.adobe.com/livedocs/flash/9.0/main/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs\_Parts&file=00001136.html 中找到,以下的注释是对 case 语句,请注意!case 后是对应键的编码,//后是对应的键字.
在每个 case 语句里,是一个调用建立新基类模型 3D 对象的函数 */
switch (event.keyCode)
{
    case 49: //1
        initCone();
        break;
    case 50: //2
        initCube();
        break;
    case 51: //3
        initCylinder();
        break;
    case 52: //4
        initGeodesicSphere();
        break;
    case 53: //5
        initGridPlane();
        break;
    case 54: //6
        initLineSegment();
        break;
    case 55: //7
        initPlane();
        break;
    case 56: //8
        initRegularPolygon();
        break;
    case 57: //9
        initRoundedCube();
        break;
    case 47: //0
        initTorus();
        break;
    case 81: //Q
```

```

        initTriangle();
        break;
    case 87: //W
        initSeaTurtle();
        break;
    case 69: //E
        initSphere();
        break;
    case 82: //R
        initTrident();
        break;
    case 84: //T
        initSkybox();
        break;
    case 89: //Y
        initSkybox6();
        break;
    default:
        initSphere();
        break;
    }
}

```

/* 正如用 addChild() 函数把 3D 对象添加到场景里,使之在场景里可看见一样,为从场景里删除它,调用 removeChild() 函数。

在一个新的原始 3D 对象建立之前,由 onKeyUp() 函数调用 removeCurrentPrimitive() 函数,它删除当前显示的 3D 对象,其代码如下 */

```

protected function removeCurrentPrimitive():void
{
    scene.removeChild(currentPrimitive);
    currentPrimitive=null;
}
}

```

PrimitivesDemo 类的余下部分是由各个函数组成的,通过本程序展示建立,显示各个原始 3D 对象函数。

这些函数的详细介绍,在后面的各节里展示。除此以外还列出了一些参数,这些参数提供用于原始 3D 类的构造函数。

正如在第 1 章中所见到的,把包含在 Away3D 库中的类,提供给初始化对象,作为其构造函数的参数。建立初始化对象通常用字母符号法,这是共同的习惯作法;但也有例外,如即将看到的 Trident、Skybox 和 SkyBox6 这些类。

除非另外指出,下面的函数参数列表中出现的参数,都是相对于初始化对象的属性。

在相应的地方,也提供了用线框模型展示的这些伴随原始 3D 类的图像,并且在它上面

用的是 Bitmap 位图格式进行材质的纹理映射。请注意,这些图像仅用于作例证的目的,并不反映例子代码的输出,例子代码产生的原始 3D 对象,用的是默认的 WireColorMaterial 的材质。

以下展示的所有基类模型,除了 Trident 类,其余的类都是直接或间接继承了 Mesh 类,例如三角形 triangle、海龟 sea turtle、线段 line segment 和天空盒 Skybox 等形状类是直接继承了 Mesh 类,而其余的是继承了 AbstractPrimitive 类,而这些类又转而继承了 Mesh 类。

利用类的构造函数提供初始化对象时,要向前传送到 Mesh 的构造函数。这表明,有很多初始化对象的参数对所有基类模型是共同的(排除 Trident 类,因为它并不继承 Mesh 类。还要排除 Skyboxe,因为它不将初始化对象传递到 Mesh 类的下面)。

这些参数的主要部分,牵涉到如何应用材质,有关材质应用的细节,请参见第 5 章。

出现在下面的所有表中用于初始化对象的大多数参数,也是一旦它们在实例化时需要设置或访问的属性。因此所有类对象初始化的共同属性设置为:

```
var plane=new Plane( {bothsides:true});
```

也可这样写:

```
var plane=new Plane();  
plane.Bothsides=true
```

表 2-1 列出了所有类对象初始化可以接受的共同属性(参数)。

表 2-1 用于初始化对象的共同参数

参数	数据类型	默认值	说 明
outline	Material	null	定义一线段材质,用于 3D 对象的输出
material	Material	null	定义材质,用于显示组成网眼的底部元素
back	Material	null	定义材质,用于显示组成网眼的底部元素的表面
bothsides	Boolean	false	指出前面和背面是否应该都提供,为 True,剔除背面

下面分别讲述各种具体模型对象的初始化代码及相应的参数设置。

1. 圆锥体

建立一个新原始类的实例,是非常简单的。一旦相应的类从 away3d.primitives 包引入,一个新的类实例便建立起来,并能直接添加到场景。这和三角面、Sprite3D 和 Segment 等类的对象不同,这些类的对象首先要添加到 Mesh。

事实上,手工建立三角面所遵循的过程,与原始类所用的过程是非常相似的。其最大的区别就是,原始类全都继承 Mesh 类(除 Trident、Skybox 和 SkyBox6 类之外),把三角面自己本身添加到场景,而不是把它们添加到各自的 Mesh 类的实例。

Cone 类对象的图形表示如图 2-10 所示。

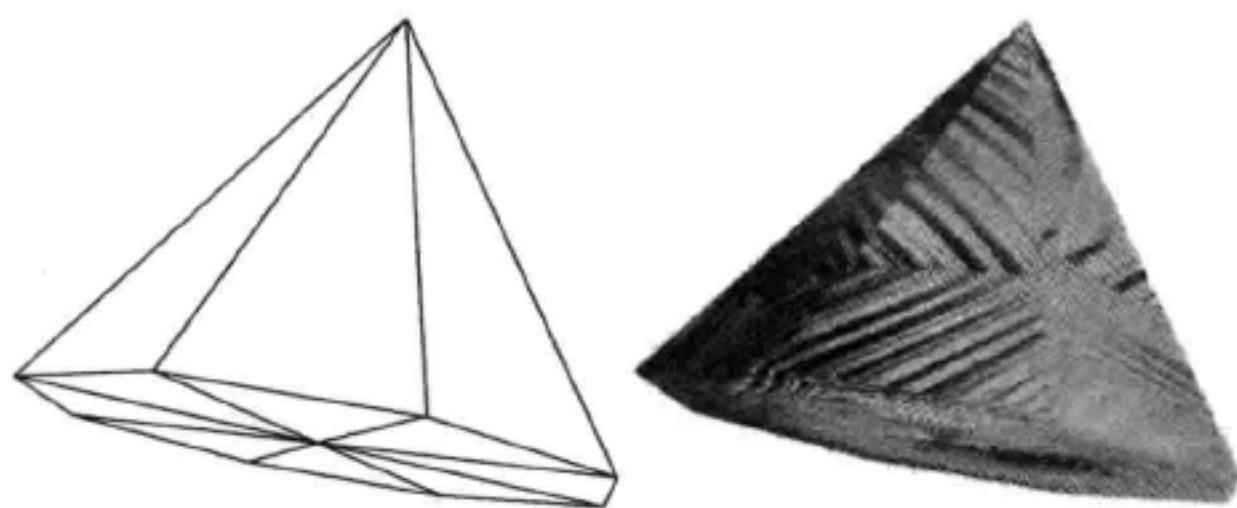


图 2-10 Cone 类对象的图形表示

initCone() 函数用于建立并显示一个圆锥体 Cone 类的实例。

```
protected function initCone():void
{
    currentPrimitive=new Cone(
    {
        Height:150
    }
    );
    scene.addChild(currentPrimitive);
}
```

表 2-2 列出了圆锥体类 Cone 对象初始化可能所需的参数。

表 2-2 Cone 类对象的初始化参数

参数	数据类型	默认值	说 明
radius	Number	100	定义圆锥体的底部半径
height	Number	200	定义圆锥体的高
segmentsW	int	8	定义组成圆锥体的水平段的数值
segmentsH	int	7	定义组成圆锥体的垂直段的数值
openEnded	Boolean	false	定义组成圆锥体的末端是否打开,或关闭
yUp	Boolean	true	确定圆锥体是否应该沿 Y 轴方向向上

2. 立方体

Cube 类建立一个标准的六面体,默认情况下,朝向的这些面从外面是可看到的。如果想把照相机放到这立方体的里面,看见这些里面的面,可以把初始化对象的 flip 参数值设置为 True,这将反组成立方体的三角面的朝向。

图 2-11 为 Cube 类对象的图形表示。

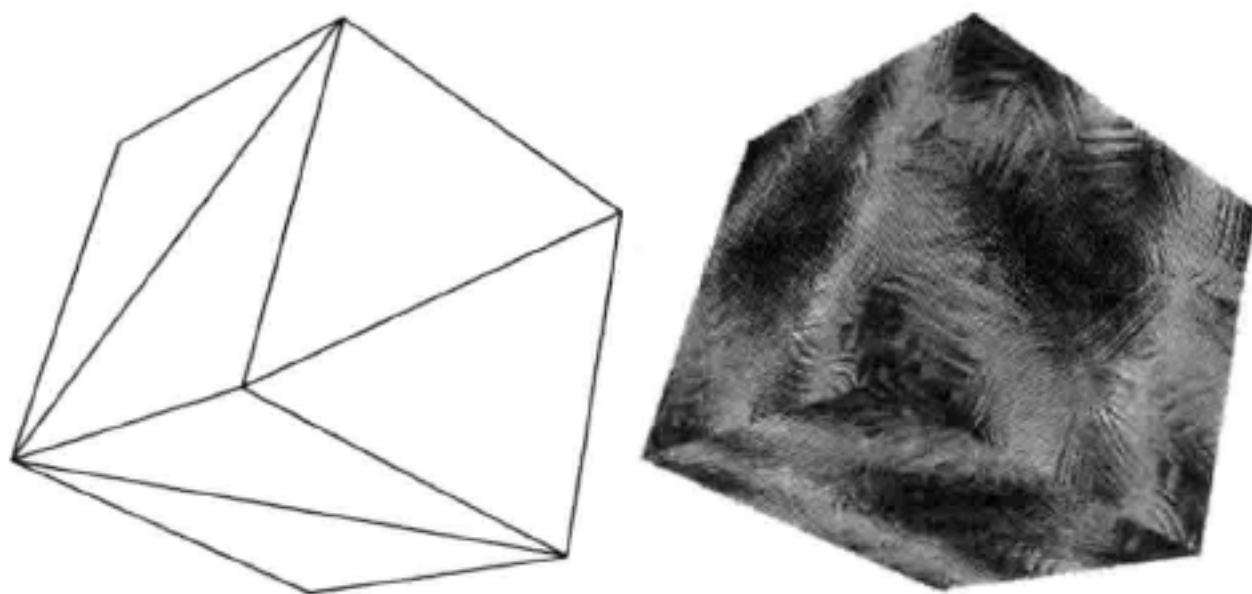


图 2-11 Cube 类对象的图形表示

initCube()函数用于建立并显示一个立方体 Cube 类的实例。

```
protected function initCube():void
{
    currentPrimitive=new Cube();
    scene.addChild(currentPrimitive);
}
```

表 2-3 中列出了立方体 Cube 类对象初始化可能所需的参数。

表 2-3 Cube 类对象的初始化参数

参数	数据类型	默认值	说 明
width	Number	100	定义立方体的宽
height	Number	100	定义立方体的高
depth	Number	100	定义立方体的深
flip	Boolean	false	反转立方体的朝向,用于使立方体从这面是可见的
segmentsW	int	1	定义组成立方体沿它的宽度方向的段的数量
segmentsH	int	1	定义组成立方体沿它的高度方向的段的数量
segmentsD	int	1	定义组成立方体沿它的深度方向的段的数量
mappingType	String	CubeMappingType NORMAL/"normal"	定义 UV 坐标如何应用于立方体。有效值是 normal 和 map6, 这些串也定义在 CubeMappingType 类作为它的常数 NORMAL 或 MAP6
Faces	CubeMaterialsData	Null	数据结构,保存立方体的 6 个面的各种材质
cubeMaterials	CubeMaterialsData	Null	与表面属性相同,如果表面属性被设置, cubeMaterials 属性将被忽略

Away3D 中大多数的原始类都被设计为使用单一的材质应用到模型的表面。立方体类稍有不同,它使用 cubeMaterials 或 faces 作参数,两者都能接收 cubeMaterialsData 对象,可

指定 6 个独特的材质,应用于立方体的各个面上。cubeMaterialsData 类在 away3d.primitives.data 包里。

cubeMaterialsData 对象的构造函数要取得一定数量的初始化对象参数: front, back, left, right, top 和 bottom, 这些参数的每一个接受一种材质, 材质应用到与其相应的立方体面上。

```
new Cube(
{
    cubeMaterials: new CubeMaterialsData;
    {
        left: new BitmapFileMaterial("one.jpg"),
        front: new BitmapFileMaterial("two.jpg"),
        right: new BitmapFileMaterial("three.jpg"),
        back: new BitmapFileMaterial("four.jpg"),
        top: new BitmapFileMaterial("five.jpg"),
        bottom: new BitmapFileMaterial("six.jpg")
    }
});
```

除此之外,设置 mappingType 初始化对象参数为 map6 或 CubeMappingType. MAP6, 立方体可把分开的两行三列的纹理显示出来,这 6 个中的每一个应用到立方体的每一个面上。

这样的纹理图的例子如图 2-12 所示。

然后,将纹理图应用到立方体,代码是:

```
new Cube(
{
    mappingType: CubeMappingType. MAP6,
    material: new BitmapFileMaterial("map6.jpg")
});
```

图 2-13 显示了这两者的运行结果。

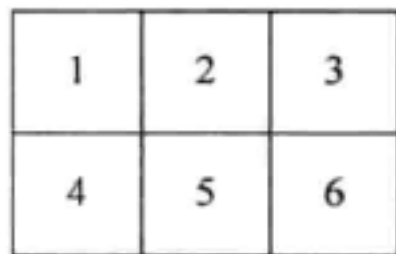


图 2-12 立方体纹理映射图示例

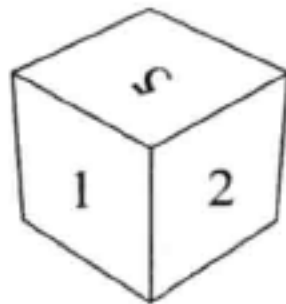


图 2-13 立方体的贴图效果

3. 柱面

Cylinder 类可建立一个实心圆柱体,也可建立一个圆柱筒的 3D 对象,如图 2-14 所示。

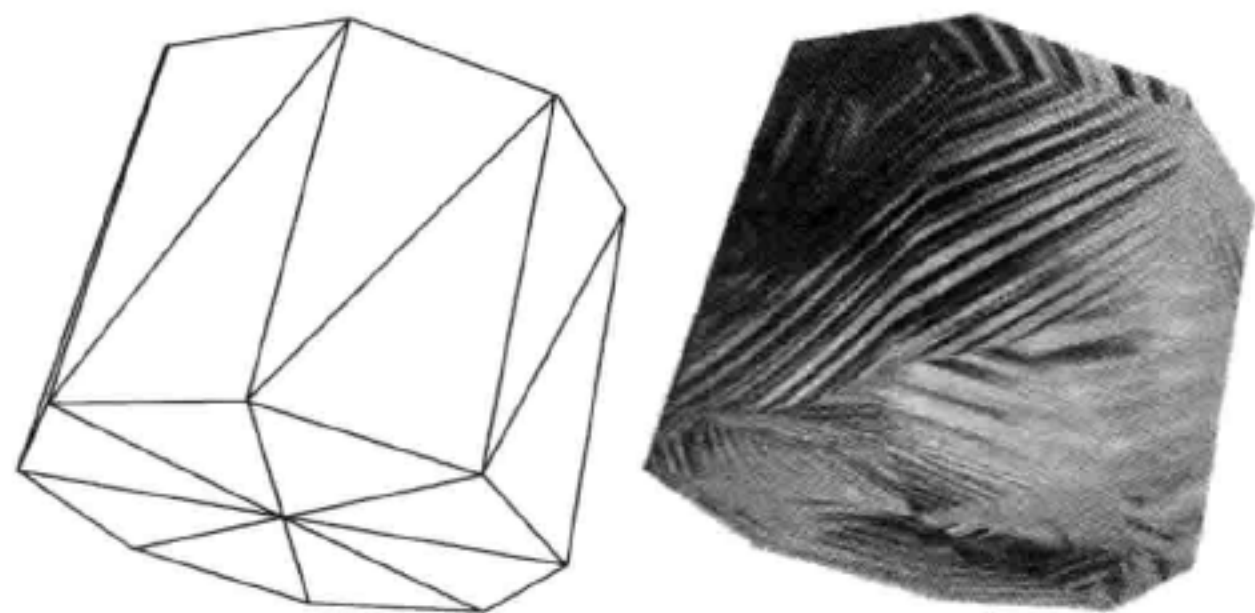


图 2-14 Cylinder 类对象的图形表示

initCylinder()函数用于建立并显示 Cylinder 实例。

```
class protected function initCylinder():void
{
    currentPrimitive=new Cylinder(
        {
            height:150
        }
    );
    scene.addChild(currentPrimitive);
}
```

表 2-4 列出了圆柱体类 Cylinder 对象初始化可能所需的参数。

表 2-4 Cylinder 类对象的初始化参数

参数	数据类型	默认值	说 明
radius	Number	100	定义圆柱体的半径
height	Number	200	定义圆柱体的高
segmentsW	int	8	定义组成圆柱体水平的段的数量,增加数量,产生更圆的圆柱体
segmentsH	int	1	定义组成圆柱体垂直段的数量
openEnded	Boolean	false	定义圆柱体的末端是否打开,还是关闭
yUp	Boolean	true	确定圆柱体是否应该沿 Y 轴的向上方向

4. 几何球面

Away3D 有两个基类模型球,这里的几何球面是用大小粗略相等的三角面构成的,更均

匀的是第二个球体。几何球面不像正式的几何球体,组成正式的几何球体的朝向顶或底的三角面更小些。

图 2-15 是一个几何球面的例子。

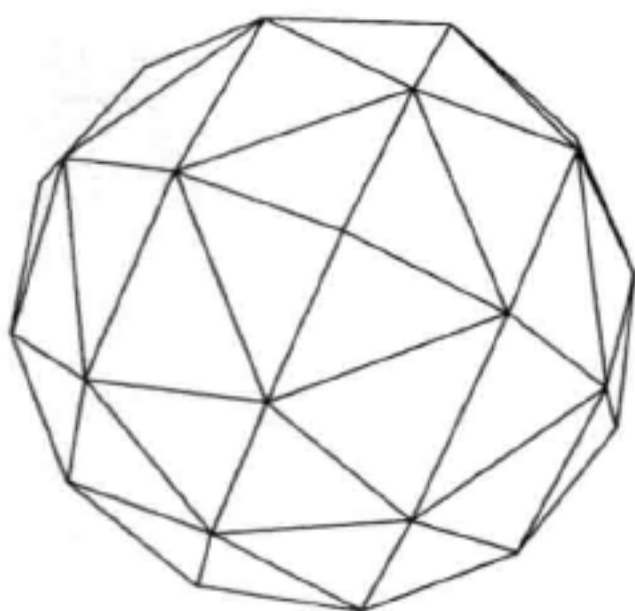


图 2-15 几何球面

把几何球面与如图 2-16 所示的正式的几何球体相比较。注意:构成球体的朝向底部和顶部的这些三角形,比起围绕中间的三角形要小一些。

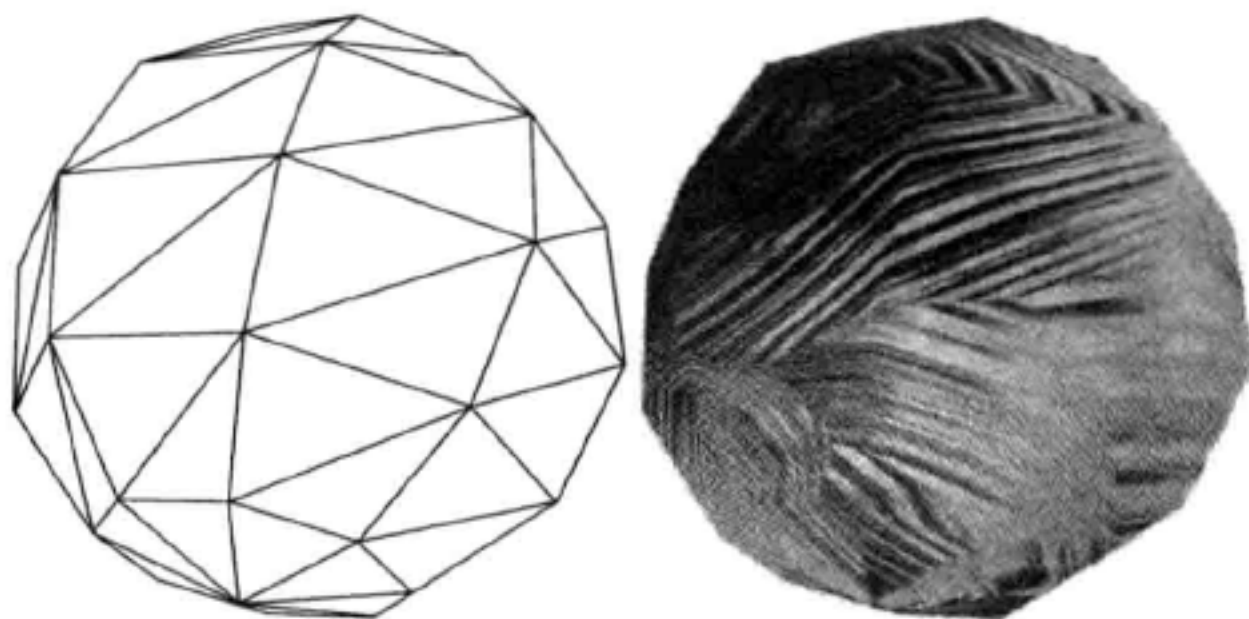


图 2-16 几何球体

几何球面与标准球体相比,在使用相同数量的三角面的情况下,几何球面能生成更圆的球面。

因为把 UV 坐标指定给几何球面的这种方法,在显示 Bitmap 材质时并不是非常有用的。有关原始球的更多信息,请参见 2.3 节。

initGeodesicSphere()函数,用于建立并显示几何球面类的实例。

```
protected function initGeodesicSphere():void  
{
```



```
currentPrimitive=new GeodesicSphere();
scene.addChild(currentPrimitive);
}
```

表 2-5 列出了几何球类 GeodesicSphere 对象初始化可能所需的参数。

表 2-5 GeodesicSphere 类对象的初始化参数

参数	数据类型	默认值	说 明
radius	Number	100	定义几何球的半径
fractures	int	2	定义三角几何的级别,级别越高产生越光滑,细节越丰富的球

5. 网格平面

网格平面即 GridPlane 类对象,是一个有很多矩形的格子,它是一个判断其他 3D 对象在场景里位置的方便的工具。与三叉戟原始组件相结合,可以显示场景里的坐标。它可以很容易复制出简单 3D 模型的应用例子。当看见图 2-17 的时候,网格平面允许立即看到 3D 对象相对于原点的位置,在应用程序的调试阶段,这是非常有用的。

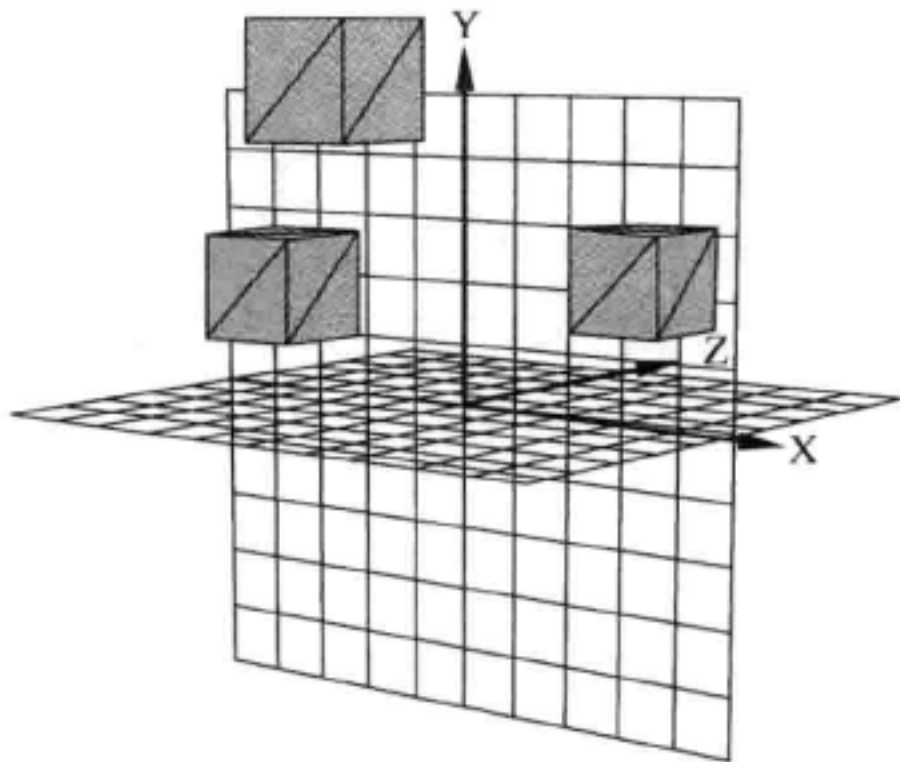


图 2-17 屏幕截图

构造网格平面用的是一些段,而不是三角面,这就允许它显示矩形,而不是三角形,当有一个线框材质应用到它时,使用三角面的结构被显示为平面的基类模型。

initGridPlane()函数用于建立并显示 GridPlane 类的实例。

```
protected function initGridPlane():void
{
    currentPrimitive=new GridPlane(
    {
        segment:4
```

```

    }
    );
    scene.addChild(currentPrimitive);
}

```

表 2-6 列出了网格平面类 GridPlane 对象初始化可能所需的参数。

表 2-6 GridPlane 类对象的初始化参数

参数	数据类型	默认值	说 明
widths	Number	100	定义栅格宽度
height	Number	100	定义栅格高度
segments	int	1	设置栅格每边分割的数量,如果值为 2,将使栅格里有 4 个矩形
segmentsW	int	1	定义组成栅格水平的段的数量,这个属性的默认值是指派给段属性的值
segmentsH	int	1	定义组成栅格垂直段的数量,这个属性的默认值是指派给段属性的值
yUp	Boolean	true	确定栅格平面是否应该沿 Y 轴的向上方向

6. 线段

LineSegment 类是另一个用段而不是用三角面建立基类模型的例子,它能显示在空间里两点之间的直线,如图 2-18 所示。为方便起见,可以用 LineSegment 类建立网格 mesh,然后用手工把段再添加到线段。就像在 3D 对象的基本元素中这一节中所介绍的那样。



图 2-18 空间中两点之间的直线表示

initLineSegment() 函数用于建立并显示线段类的实例。

```

protected function initLineSegment():void
{
    currentPrimitive=new LineSegment(
    {

```

```
        edge:500
    }
    );
    scene.addChild(currentPrimitive);
}
```

表 2-7 列出了线段类 LineSegment 对象初始化可能所需的参数。

表 2-7 LineSegment 类对象的初始化参数

参数	数据类型	默认值	说 明
edge	Number	100	设置默认的线段,起点到终点间沿 X 轴方向的单位数。如果值为 100,建立的线段的起点坐标是(-50,0,0),终点坐标是(50,0,0)
start	Vector3D	(-edge/2,0,0)	设置线段起点,如果指定,覆盖原默认值
end	Vector3D	(edge/2,0,0)	设置线段终点,如果指定,覆盖原默认值
segments	Number	1	设置线段的数量

7. 平面

平面在默认的情况下是一个矩形,它的一面是可见的,当从后面观察的时候,背面是排除处理(用于提高 Away3D 的运行性能,不画出三角面的背面),防止基类模型被绘制,如图 2-19 所示。将初始化参数 bothsides 的值设置为 true,并重载这种行为,可确保平面从前面以及从背面都是可看到的。

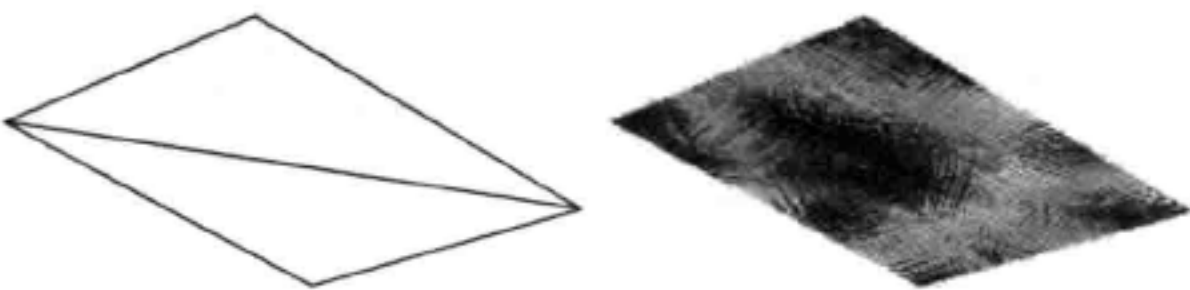


图 2-19 平面

initPlane()函数用于建立并显示平面类的实例。

```
protected function initPlane():void
{
    currentPrimitive=new Plane(
    {
        bothside:true
    }
    );
    scene.addChild(currentPrimitive);
}
```


表 2-8 列出了平面类 Plane 对象初始化可能所需的参数。

表 2-8 Plane 类对象的初始化参数

参数	数据类型	默认值	说 明
width	Number	100	设置平面宽度
height	Number	100	设置平面高度
segmentsW	int	1	设置每边段的数量,如果值为 2,则平面由 4 段组成,该属性默认值传到段的属性值
segmentsH	int	1	定义组成平面垂直段的数量,该属性默认值传到段的属性值
yUp	Boolean	true	确定平面是否应该沿 Y 轴的向上方向

8. 规则多边形

三角形 Triangle 类创建有三条边的原始对象,平面 Plane 类创建带有四条边的原始对象。而规则多边形 RegularPolygon 类则比 Triangle 类和 Plane 类稍微灵活,能够创建有任意条边(大于三条边)的规则多边形。

规则多边形 RegularPolygon 类与平面 Plane 类一样,除非把它的初始化对象的 bothsides 参数设置为 true,否则从背面是看不到的。

图 2-20 为规则多边形示例。

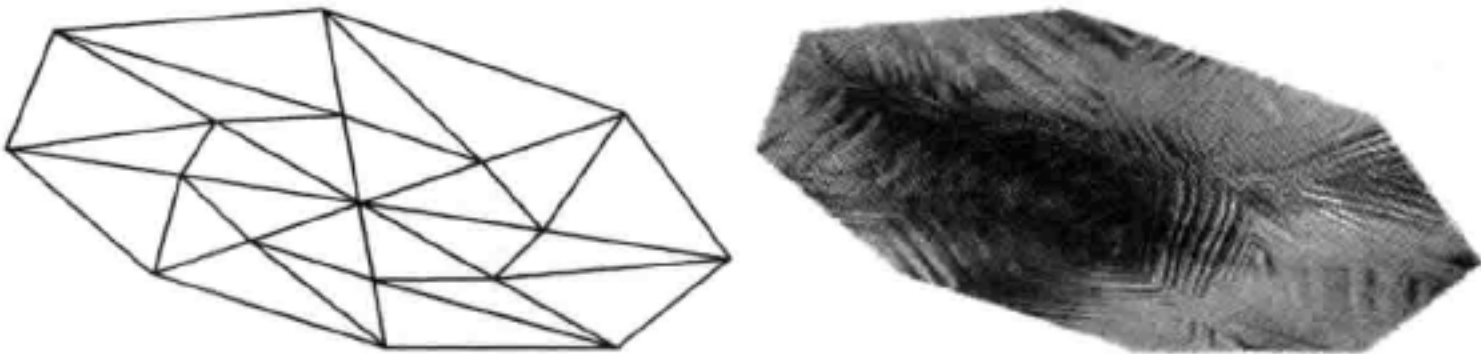


图 2-20 规则多边形

initRegularPolygon()函数用于建立并显示正多边形类的实例。

```
protected function initRegularPolygon():void
{
    currentPrimitive=new RegularPolygon(
    {
        bothside: true
    }
    );
    scene.addChild(currentPrimitive);
}
```

表 2-9 列出了规则多边形类 RegularPolygon 对象初始化可能所需的参数。

表 2-9 RegularPolygon 类对象的初始化参数

参数	数据类型	默认值	说 明
radius	Number	100	定义多边形的直径
sides	int	8	定义多边形的边数
subdivision	int	1	定义多边形的细分数,较大值增加了多边形的三角数
yUp	Boolean	true	如果为 true,建立的多边形在 X/Z 平面上,为 false,建立的多边形在 X/Y 平面上

9. 圆角立方体

RoundedCube 类能产生一个圆角边缘的立方体。显然,圆角立方体 RoundedCube 类比立方体 Cube 类使用了更多的三角面,因此,在需要看到圆角边缘的情况下,应该考虑采用 RoundedCube 类。圆角立方体与规则立方体从远处看是非常相似的。但是,构成圆角立方体需要增加的三角面也成指数倍增加。

图 2-21 为圆角立方体示例。

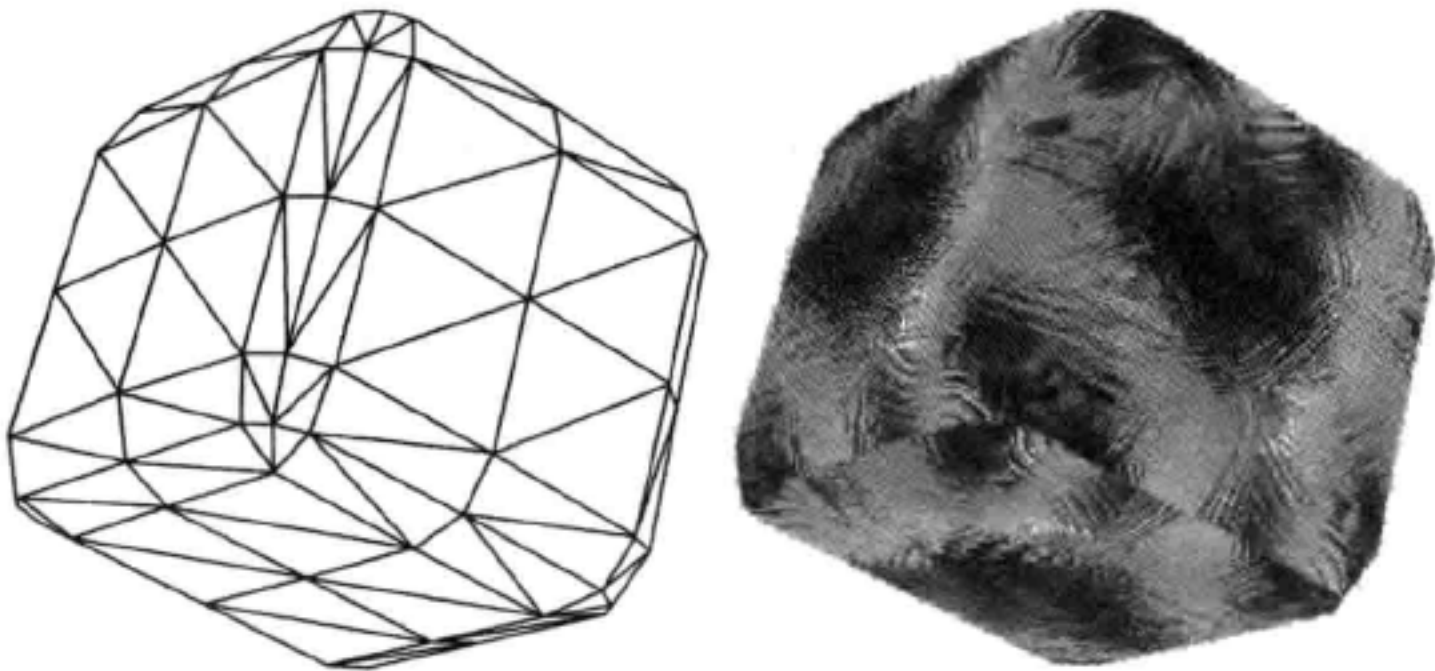


图 2-21 圆角立方体

initRoundedCube() 函数用于建立并显示圆角立方体类的实例。

```
protected function initRoundedCube():void
{
    currentPrimitive=new RoundedCube();
    scene.addChild(currentPrimitive);
}
```

表 2-10 列出了圆角正方体类 RoundedCube 对象初始化可能所需的参数。

表 2-10 RoundedCube 类对象的初始化参数

参数	数据类型	默认值	说 明
width	Number	100	定义立方体宽度
height	Number	100	定义立方体高度
depth	Number	100	定义立方体深度
radius	Number	height/3	定义立方体各角的半径
subdivision	int	2	定义圆角立方体的几何细分数
cubicmapping	Boolean	false	定义是否投影纹理到整个立方体或是沿半径调整每面的深度
faces	CubeMaterialsData	Null	保存 6 种材质的数据结构,立方体每面一种
cbeMaterials	CubeMaterialsData	Null	与 faces 参数相同(如果指定 faces 参数,该值被忽略)

10. 海龟

实际上不能认为海龟是基类模型,但是和其他基类模型一样,也能被创建和使用。Away3D 内核开发者之一 Rob Bateman 在他的演示程序中创建了大量的海龟模型特征,他的个人网站是: <http://www.infiniteturtles.co.uk/blog>(这个网址毫无疑问与 Away3D 包含的海龟这一基类模型是密切相关的)。

其他基类对象模型都各自拥有一个可以通过编程方法产生的图形形状,而海龟 SeaTurtle 类是一个复杂的 3D 模型,它可以被导入 ActionScript 文件中,第 6 章中将讲述 Away3D 能支持的各种模型格式,并详细介绍它们如何被导入 ActionScript 文件中。

如图 2-22 所示的纹理,能够在 Away3D 网站上找到,可把此图用 Photoshop 保存为名为 seaturtle.jpg 的文件,提供给应用程序当作纹理图形文件。

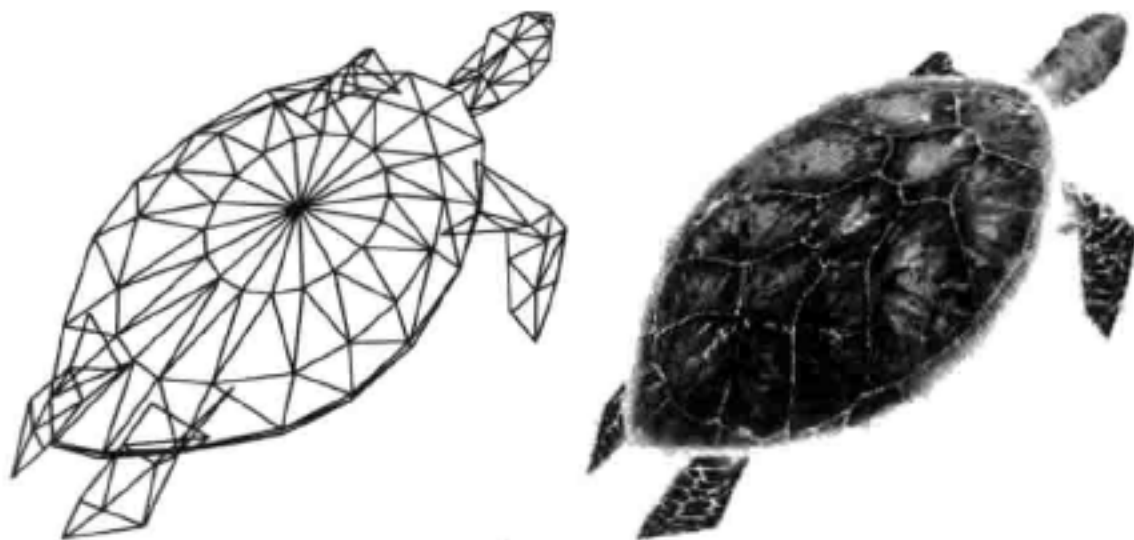


图 2-22 小海龟

initSeaTurtle()函数用于建立并显示海龟类的实例。

```
protected function initSeaTurtle():void
{
    currentPrimitive=new SeaTurtle(
    {
        Scale : 0.3
    }
    );
    scene.addChild(currentPrimitive);
}
```

SeaJurtle 类没有对象初始化所需要的参数。

Scale 初始化对象参数,在这里用于按比例均匀地缩小 3D 对象的大小, Scale 初始化对象参数由 Object3D 类来解释,SeaTurtle 类继承了 Object3D 类。第 3 章中将对 Scaling 有详细的叙述。

11. 天空盒子

Skybox 类可以建立一个巨大的、各面朝向内部的立方体。天空盒子的容积是: $800\,000 \times 800\,000 \times 800\,000$ 单位,与立方体 Cube 默认的容积 $100 \times 100 \times 100$ 相比要大很多。天空盒子设计成能装入整个场景,包括照相机和通常应用到天空盒的一个全景式视野的材质,用以显示场景以外世界的图像。

图 2-23 中透视立方体的后面显示了两个从外表看到的天空盒子的样子。通常场景中的照相机和所有其他 3D 对象全都被装在天空盒子里。但是从天空盒子外面,可得到一个有 6 个面的立方体,这个立方体装入并封套了整个场景。

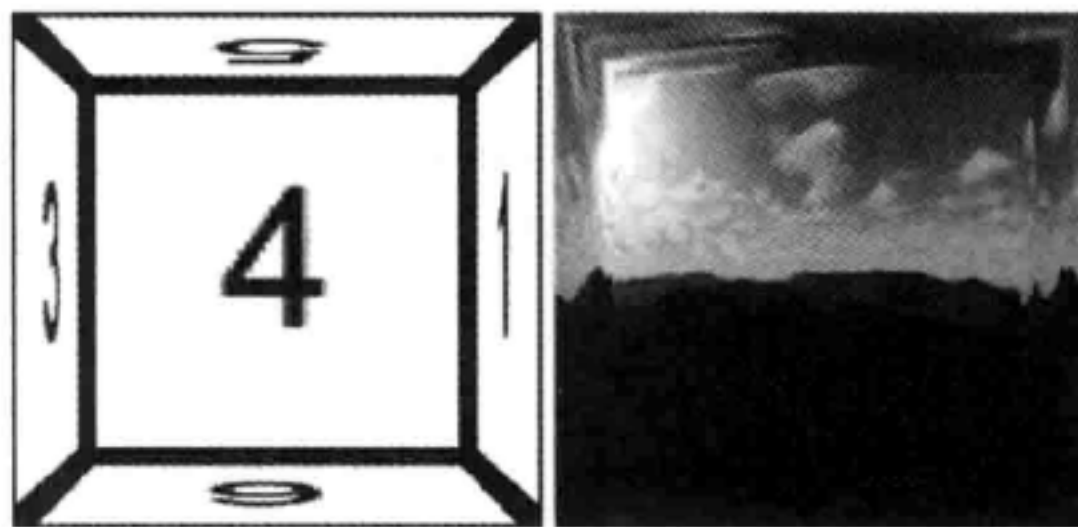


图 2-23 天空盒子

图 2-23 左边的天空盒子,有 1~6 数字表示的 Bitmap 材质应用到它的各个表面,这就很容易使人看到,材质传递到天空盒子的构造函数,结果最终又是如何映射出来的。图 2-23

右边的天空盒子,有些专门格式的天空纹理应用到它的上面,这就是在 Away3D 应用程序中实际看到的天空。

initSkybox()函数用于建立并显示天空盒子类的实例。

```
protected function initSkybox():void
{
    currentPrimitive=new Skybox(
        new BitmapFileMaterial("two.jpg"),
        new BitmapFileMaterial("one.jpg"),
        new BitmapFileMaterial("four.jpg"),
        new BitmapFileMaterial("three.jpg"),
        new BitmapFileMaterial("five.jpg"),
        new BitmapFileMaterial("six.jpg")
    );
    scene.addChild(currentPrimitive);
}
```

天空盒子类没有初始化的对象,相反,它具有 6 个参数,每个参数都定义了一个显示在立方体各表面上的材质属性,这些参数见表 2-11。

表 2-11 Skybox 类构造函数的参数

参数	数据类型	默认值	说 明
front	Material		材质用于天空盒子的前面
left	Material		材质用于天空盒子的左面
back	Material		材质用于天空盒子的背面
right	Material		材质用于天空盒子的右面
up	Material		材质用于天空盒子的上面
down	Material		材质用于天空盒子的下面

12. 天空盒子 6

Skybox6 类用于建立一个天空盒子,与 Skybox 类的功能类似,唯一的不同是它使用一个分为两行三列的材质(当 mappingType 参数设置为 map6 时的情况下,很像立方体类),然后 6 个段分别应用到立方体的 6 个面。

图 2-24 是一个 Skybox6 类的纹理例子。

1	2	3
4	5	6

图 2-24 Skybox6 类的纹理例子

initSkybox6()函数用于建立并显示 Skybox6 类的实例。

```
protected function initSkybox6():void
{
    currentPrimitive=new Skybox6(
    new BitmapFileMaterial("map6.jpg")
    );
    scene.addChild(currentPrimitive);
}
```

Skybox6 类也没有初始化的对象,它用单个参数定义显示在立方体上的材质,见表 2-12。

表 2-12 Skybox6 类构造函数的参数

参数	数据类型	默认值	说 明
material	Material		material 用在 Skybox6 上

13. 球体

Sphere 类是能够用于建立 3D 球体对象的第二种类,图 2-25 为一个球体示例。

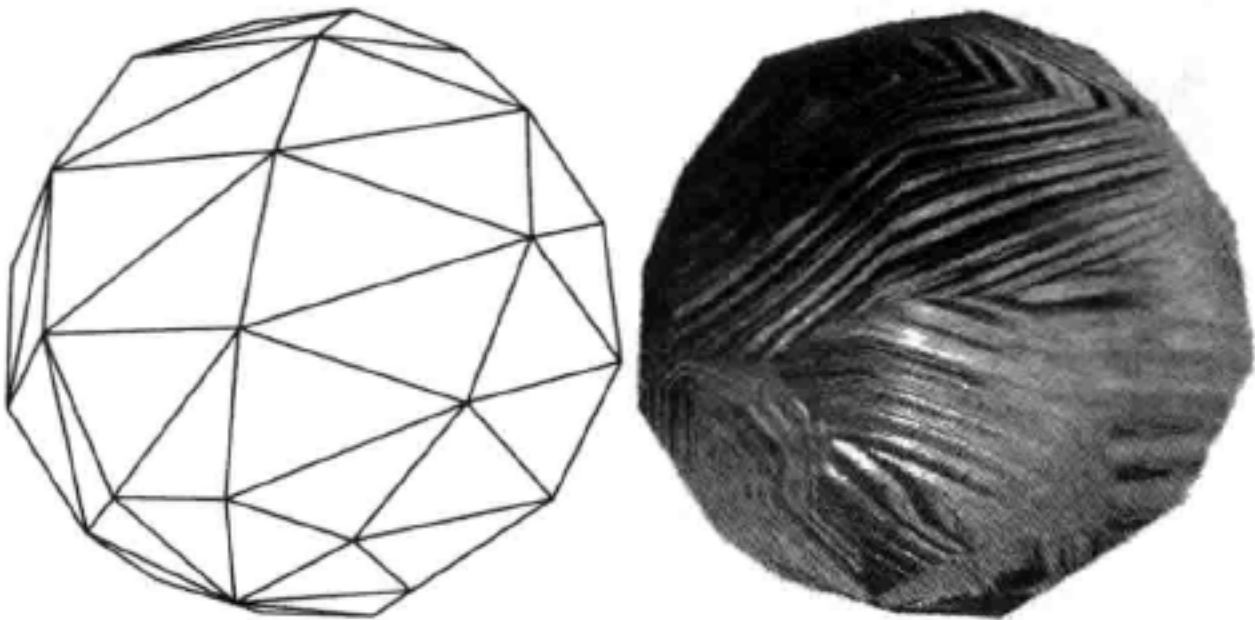


图 2-25 球体

initSphere()函数用于建立并显示球体类的实例。

```
protected function initSphere():void
{
    currentPrimitive=new Sphere();
    scene.addChild(currentPrimitive);
}
```

表 2-13 列出了球体类 Sphere 对象初始化可能所需的参数。

表 2-13 Sphere 类对象的初始化参数

参数	数据类型	默认值	说 明
radius	Number	100	定义球体的半径
segmentsW	int	8	定义组成球体水平段的数量
segmentsH	int	6	定义组成球体垂直段的数量
yUp	Boolean	true	确定三角是否应该沿 Y 轴的向上方向

既然几何球面 GeodesicSphere 类能产生比球体 Sphere 类更均匀、更圆的球,为什么还用 Sphere 类? 这个答案是很显然的,将 Bitmap 材质应用到这两个由 3D 对象建立的球体类时,比较它们的显示效果,答案就出来了。图 2-26 是由 3D 对象球体 Sphere 类建立的球体,正如所看到的,材质几乎覆盖了球的表面。

请使用相同的材质,应用到由 3D 对象几何球面 GeodesicSphere 类建立的球体上并进行比较,如图 2-27 所示。

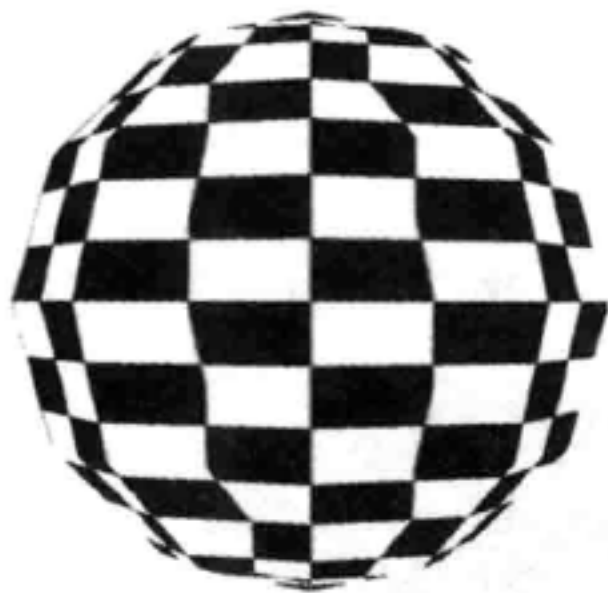


图 2-26 Sphere 类建立的球体



图 2-27 GeodesicSphere 类建立的球体

这就很清楚地说明,当几何球面 GeodesicSphere 类产生一个高质量的网格时,它的 UV 坐标有点混乱。而另一方面,球体 Sphere 类在材质方面的应用是更一致和更可用的。

然而,这仅是使用 Bitmap 材质的情况,如果使用简单材质,如 WireframeMaterial、WireColorMaterial 或 ColorMaterial,则选几何球面还是会好一些。

14. 圆环体

Torus 类可以建立一个 3D 圆形环体对象,见图 2-28。

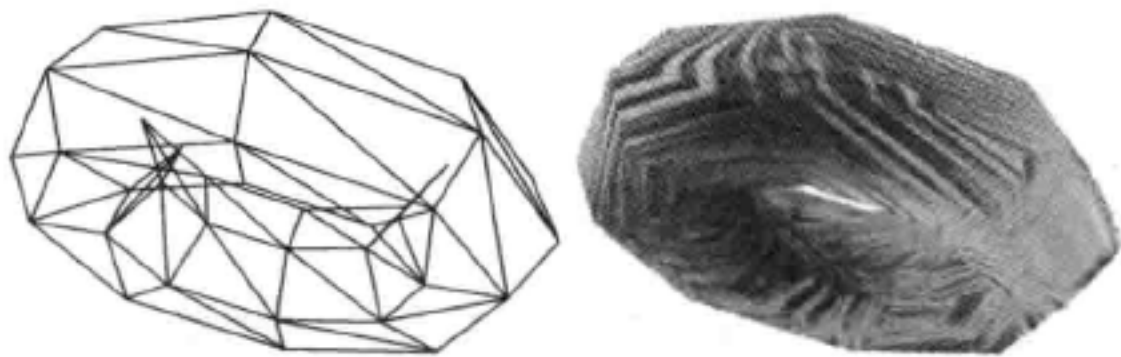


图 2-28 圆形环体

initTorus()函数用于建立并显示圆环体类的实例。

```
protected function initTorus():void
{
    currentPrimitive=new Torus(
        radius: 75,
        tube: 30
    );
    scene.addChild(currentPrimitive);
}
```

表 2-14 列出了圆环体类 Torus 对象初始化可能所需的参数。

表 2-14 Torus 类对象的初始化参数

参数	数据类型	默认值	说 明
radius	Number	100	定义圆环体整体的半径
tube	Number	40	定义圆环体管的半径,该参数不能大于整体的半径
segmentsR	int	8	定义组成圆环体半径的段的数量
segmentsT	int	6	定义组成圆环体管状的段的数量
yUp	Boolean	true	如果为 true,建立的圆环体在 X/Z 平面上,否则建立的多边形在 X/Y 平面上

15. 三角形

Triangle 类用于建立单个的三角面。与平面 Plane 类和正多边形 RegularPolygon 类一样,三角形类 Triangle 的实例从背面是不可见的,除非初始化参数 bothsides 设置为 True。

图 2-29 为一个三角形面示例。

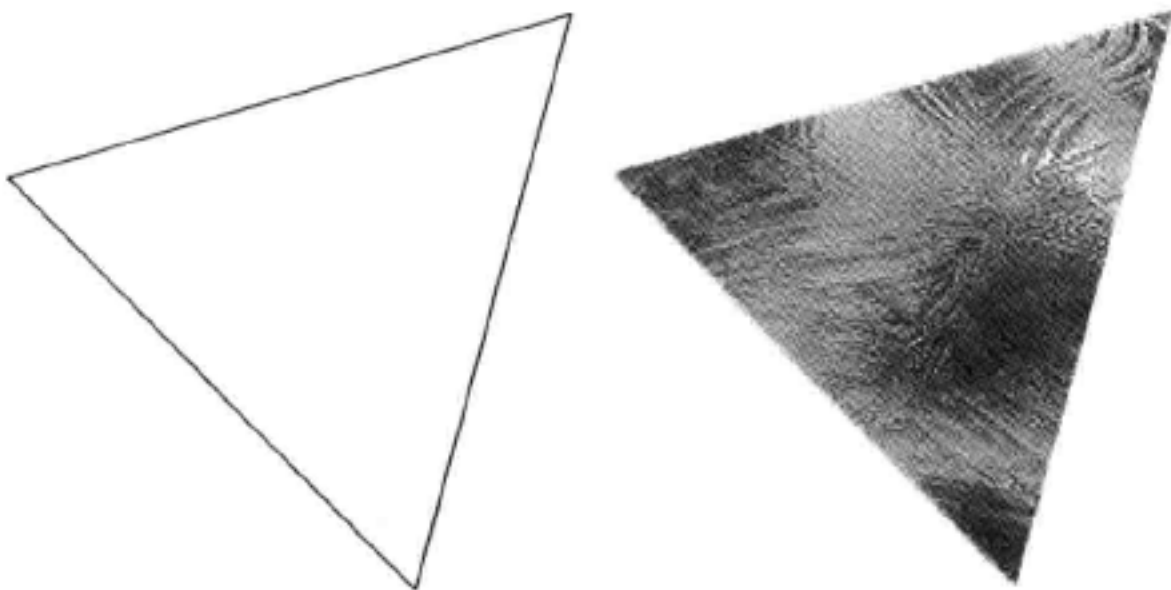


图 2-29 三角面

initTriangle()函数用于建立并显示三角形类的实例。

```
protected function initTriangle():void
{
    currentPrimitive=new Triangle(
        bothsides: true
    );
    scene.addChild(currentPrimitive);
}
```

表 2-15 列出了三角形类 Triangle 对象初始化可能所需的参数。

表 2-15 Triangle 类对象的初始化参数

参数	数据类型	默认值	说 明
edge	Number	100	设置三角形大小
yUp	Boolean	true	如果为 true,建立的圆环体在 X/Z 平面上,否则建立的多边形在 X/Y 平面上

16. 三叉戟

Trident 类,建立有三种颜色的箭头,分别代表 X、Y 和 Z 坐标轴。如果 showLetters 初始化参数设置为 true,每个坐标轴也都带有标签。这样它在调试应用程序阶段是非常有用的,因为它能显示 3D 对象的方向。第 3 章将解释 3D 对象的方向朝向会影响一些功能的效果。

图 2-30 为一个三叉戟示例。

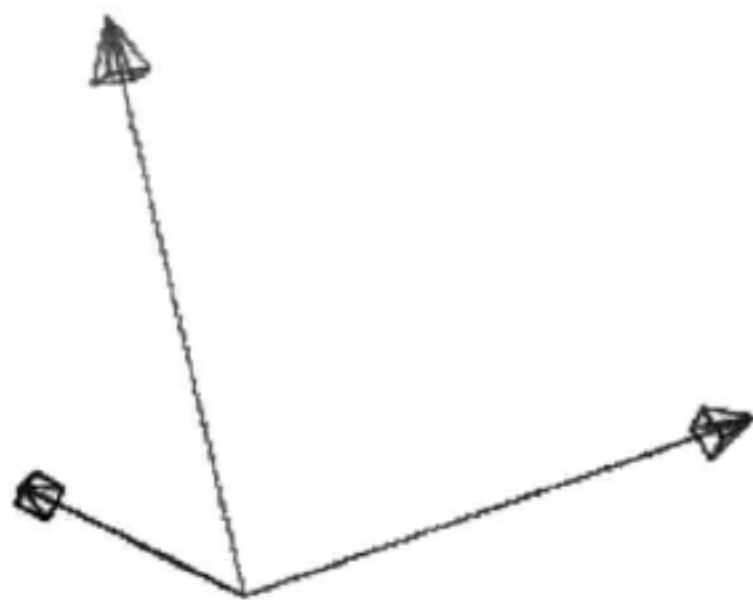


图 2-30 三叉戟

initTrident()函数用于建立并显示三叉戟类的实例。

```
protected function initTrident():void
{
    currentPrimitive=new Trident(100);
    scene.addChild(currentPrimitive);
}
```


表 2-16 列出了三叉戟类 Trident 对象初始化可能所需的参数。

表 2-16 Trident 类对象的初始化参数

参数	数据类型	默认值	说 明
len	Number	1000	定义三叉戟坐标轴长度
showLetters	Boolean	false	定义三叉戟是否显示 X、Y 和 Z 轴的标签

移动对象

场景中的 3D 对象能够用多种方法制成动画片。第 2 章已经介绍过有关这方面的内容,在那里原始模型 3D 对象是用 PrimitivesDemo 应用程序建立的,通过修改它们的 rotationX、rotationY 和 rotationZ 属性使对象旋转。移动、缩放或者旋转 3D 对象,又称为转换 3D 对象。本章将详细地探讨如何转换场景中的 3D 对象。

本章主要内容:

- 不同的坐标系统
- 通过修改 3D 对象位置,旋转和移动、转换 3D 对象
- 使用 TweenLite 库,转换 3D 对象
- 嵌套 3D 对象

3.1 全局坐标、父坐标和局部坐标

在第 1 章中,已经见到过使用 X、Y 和 Z 坐标系统,以及坐标系统里的 3D 对象是如何定位于场景里的。在 Away3D 里,从三个不同的角度将坐标分别定义为:全局坐标、父坐标和局部坐标。弄清它们之间的区别是很重要的,因为在 Away3D 工作中,所有的移动、旋转和缩放操作,都是分别相对于这三个坐标系统中的一个进行的。

3.1.1 世界空间

全局坐标系统代表朝向相对于场景原点的点或向量的坐标系统,该坐标系统也被称为

世界空间。如图 3-1 所展示的球,取自第 1 章中的例子。

下面是建立这个球的代码:

```
var sphere:Sphere=new Sphere(  
{  
    x: 0,  
    y: 0,  
    z: 500  
});
```

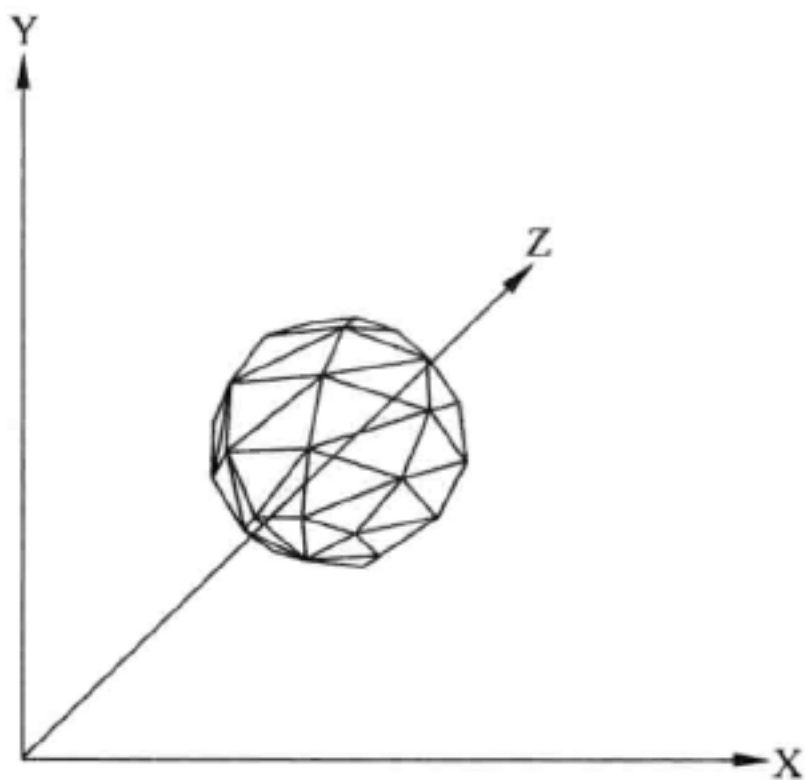


图 3-1 全局坐标系统与 3D 球

提供给 Sphere 类结构函数的初始化对象,把球设置在(0,0,500)的初始位置,这个位置是相对于 3D 对象的父容器的位置,在这样的情况下,球添加到场景里作为其子容器。

```
scene.addChild(sphere);
```

因为将球添加到场景里作为其子容器,所以球的位置是相对于场景的。这就表示在世界空间里球的位置是(0,0,500)。

3.1.2 父空间

父级坐标系统用于代表点或向量朝向相对于某一 3D 对象定位于父容器里的位置。这个坐标系统被称为父空间。

注意到在 3.1.1 节中,通过 X、Y 和 Z 轴的属性,分配给 3D 对象的位置是相对于它的父容器的位置。当 3D 对象的父容器是 Scene3D 的时候,父空间和世界空间是相同的。然而,Scene3D 并不是保存 3D 对象作为其儿子的唯一容器。Scene3D 类继承了 ObjectContainer3D 类, ObjectContainer3D 类定义了函数 addChild(), 我们曾经使用 addChild() 函数把 3D 对象添加到 Scene3D 里。这样就像 Scene3D 作为 3D 对象的容器一

样,也可使用 `ObjectContainer3D` 作为 3D 对象的容器。

下面建立一个叫做 `GroupingExample` 的新应用程序,它继承了第 1 章中的 `Away3DTemplate` 类。

```
package
{
    /* 要引入 ObjectContainer3D 类,以便在程序中是有效的 */
    import away3d.containers.ObjectContainer3D;
    Import away3d.primitives.Sphere;
    public class GroupingExample extends Away3DTemplate
    {
        public function GroupingExample()
        {
            super();
        }
    }
}
```

下面的代码重载了初始场景函数 `initScene()`,建立一个 `ObjectContainer3D` 对象,使它的位置是(0,0,500),并把它添加到场景里作为场景的“儿子”。这个容器是不可见的,即使把它加到场景里,该容器仍将是不可见的。

```
protected override function initScene():void
{
    super.initScene();
    var container:ObjectContainer3D=new ObjectContainer3D(
    {
        x: 0,
        y: 0,
        z: 500
    }
    );
    scene.addChild(container);
}
```

接下来建立一个新的球对象,但这一次设置它的位置是(0,0,0),这将把球定位在它的父级坐标的原点。

```
var sphere:Sphere=new Sphere(
{
    x: 0,
    y: 0,
    z: 0
}
);
```

/* 现在,不是把球添加到场景 scene 里作为场景的儿子,而是把它添加到容器 container 里,作为容器的儿子 */

```
    container.addChild(sphere);
  }
}
```

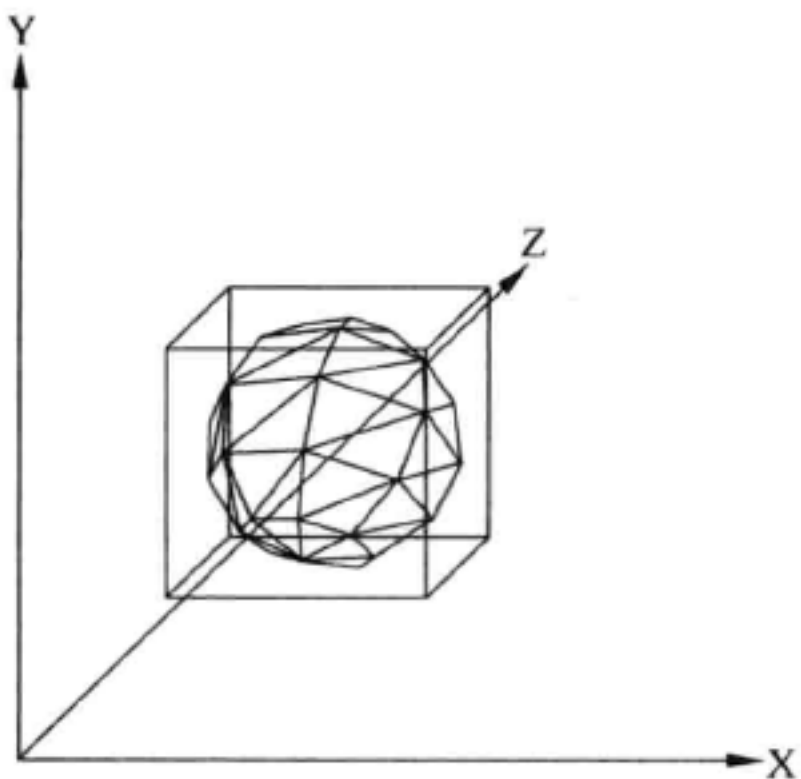


图 3-2 世界空间与父空间

图 3-2 展示了球现在在场景中的位置,父空间容器的位置是(0,0,500),由于容器的父空间是场景,所以在世界空间里容器的位置是(0,0,500)。

在父空间里球的位置是(0,0,0)。考虑到球的父容器的世界空间的位置是(0,0,500),而球是坐落在父容器的原点,所以球的世界空间的位置是(0,0,500)。图 3-2 中的立方体表示的是球的父容器。

3.1.3 局部空间

局部坐标系用于表示朝向相对于某一单个 3D 对象的点或向量的位置,该坐标系又称为局部空间。很多运动、旋转,以及缩放一个 3D 对象的操作,就是在局部空间里进行的。

为了展示这些,先建立一个名为 LocalAixsMovement 的例子。

```
package
{
    Import away3d.primitives.Sphere;
    public class LocalAixsMovement extends Away3DTemplate
    {
        public function LocalAixsMovement ()
        {
```

```

        super();
    }
    protected override function initScene():void
    {
        super.initScene();
        /* 建立了一个新的原始球体,定位,并添加到场景中 */
        var sphere:Sphere=new Sphere(
            {
                x: 0,
                y: 0,
                z: 500
            }
        );
        scene.addChild(sphere);
        /* 这时候,球坐落在沿世界空间 Z 轴 500 单位的位置,如图 3-3 所示.

```

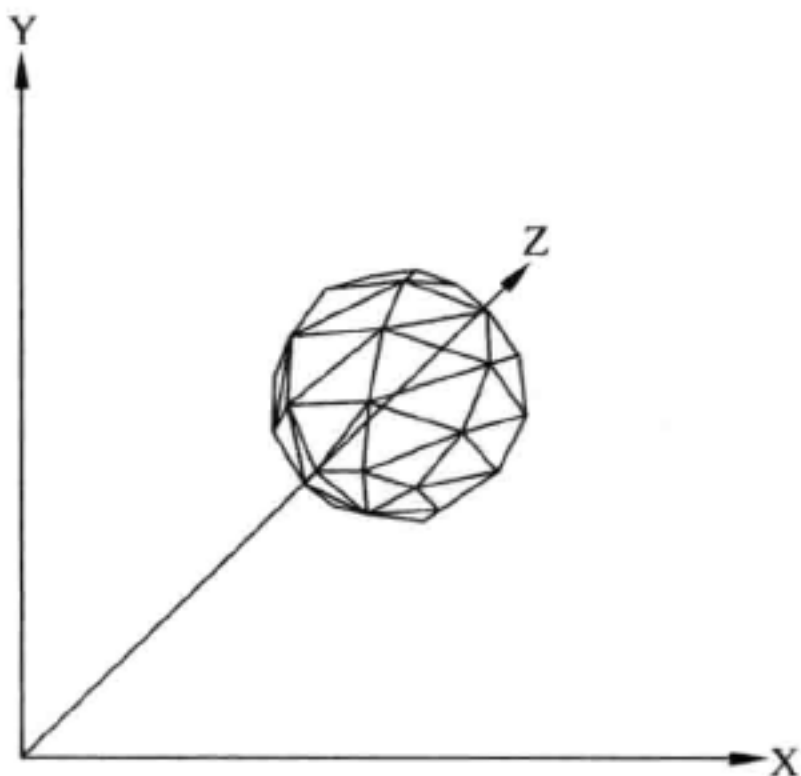


图 3-3 局部坐标与 3D 球

将球沿局部空间坐标 Y 轴旋转 -90° , 这样就修改了球的局部坐标轴的 X 和 Z 坐标的朝向 */

```

        sphere.rotationY=-90;
        /* 图 3-4 展示了球现在局部坐标系统的朝向与全局坐标系统的比较 */
        /* 向前移函数 moveForward() 将球沿它的局部坐标 Z 轴向前移 50 单位 */
        sphere.moveForward(50);
    }
}

```

球的最后位置展示在图 3-5 中。注意到球的移动是沿它的局部坐标轴的 Z 轴。在 Away3D 中, 向前的方向认为是 Z 轴的正向。

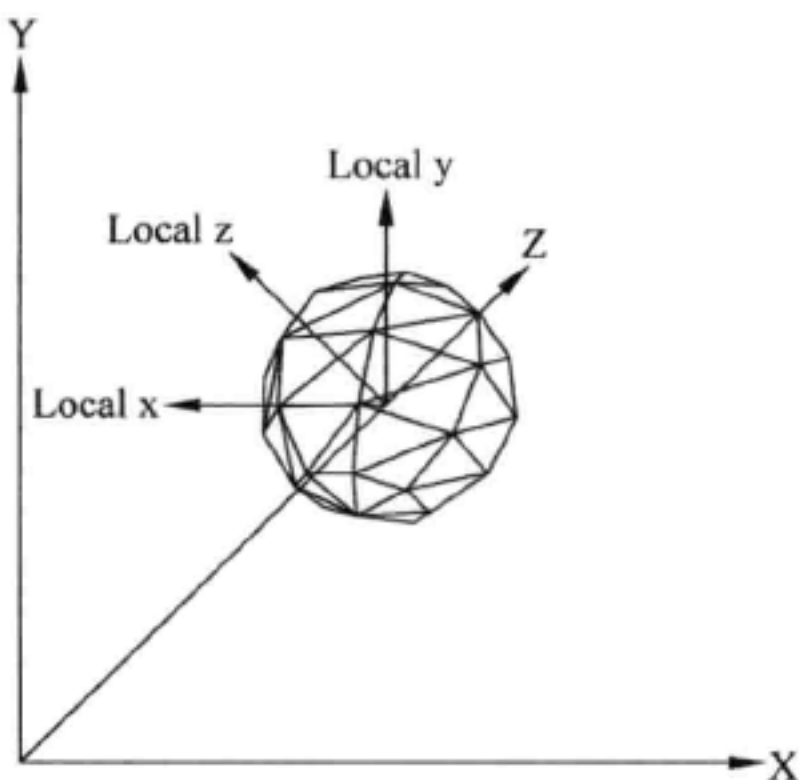


图 3-4 局部坐标(XYZ)与全局坐标(XYZ)

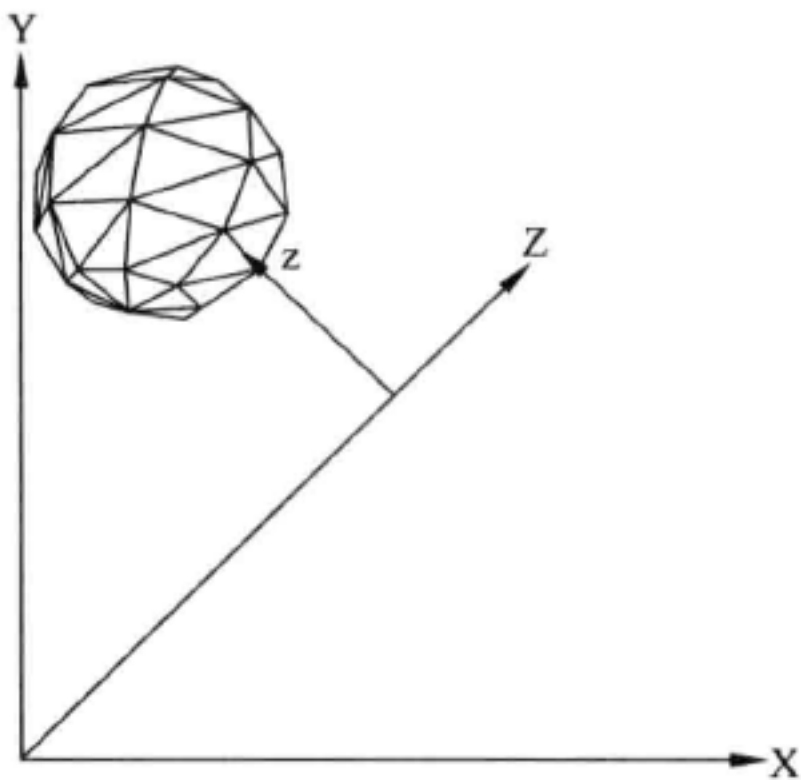


图 3-5 3D 球的移动

3.2 转换函数/属性及其坐标系统

表 3-1 列出了能用于转换 3D 对象的各个函数/属性,以及与它们相应的坐标系统。这些函数/属性都是在 Object3D 类中定义的。

表 3-1 3D 对象转换函数/属性及对应的坐标系统

函数属性	世界空间	父空间	局部空间
位置			
scenePosition	*		
x		*	

续表

函数属性	世界空间	父空间	局部空间
y		*	
z		*	
position		*	
moveForward			*
moveBackward			*
moveLeft			*
moveRight			*
moveUp			*
moveDown			*
translate			*
moveTo		*	
旋转			
scenePivotPoint	*		
pivotPoint			*
movePivot			*
rotationX		*	
rotationY		*	
rotationZ		*	
rotateTo		*	
lookAt		*	
eulers		*	
pitch			*
roll			*
yaw			*
rotate			*
缩放			
scaleX			*
scaleY			*
scaleZ			*
scale			*
转换			
sceneTransform	*		
transform		*	

3.3 修改位置

我们已经知道如何用 3D 对象的 `x`、`y` 和 `z` 的属性设置它的位置, Away3D 还包含很多附加的属性和函数, 它们也能用于 3D 对象在场景里的移动。

3.3.1 `x`、`y` 和 `z` 属性

一个 3D 对象的起时位置, 经常用初始化对象的 `x`、`y` 和 `z` 的属性来指定, 这种方法在本书的前面很多例子中已经用过了。当 3D 对象一旦建立之时, 这种方法就设置其位置属性, 而不是让它们设置为默认值, 以后再修改它们。前一种方法更有效。因此首选这种方法定义一个 3D 对象的初始位置。

```
var sphere:Sphere=new Sphere(  
{  
    x: 10,  
    y: 20,  
    z: 30  
});
```

虽然有很多 Away3D 类接受初始化对象, 如在第 2 章中所介绍的 Trident 类一样, 有几个类并没有初始值。

3D 对象的初始位置也能在它创建时, 通过指定对象自身的 `x`、`y` 和 `z` 属性值来确定。

```
var sphere:Sphere=new Sphere();  
    Sphere.x:=10;  
    Sphere.y:=20;  
    Sphere.z:=30;
```

3D 对象的位置总是设置成相对它父空间的位置, 对 `x`、`y` 和 `z` 属性和 `position` 属性是真实的。通过访问 `scenePosition` 属性, 可以找到 3D 对象在场景(或世界空间)里的位置。然而, 这些都是只读属性。

3.3.2 位置属性

位置属性能够用向量 `Vector3D` 对象来设置, 它能够一次指定 3D 对象相对于父空间里沿 `x`、`y` 和 `z` 坐标轴的位置。

```
var sphere:Sphere=new Sphere();
sphere.position=new Vector3D(10,20,30);
```

3.3.3 移动函数

Away3D 有 6 个移动函数：向前移 `moveForward()`，向后移 `moveBackward()`，向右移 `moveRight()`，向左移 `moveLeft()`，向上移 `moveUp()` 和向下移 `moveDown()`。这些函数都能够用于移动一个 3D 对象。移动是相对于 3D 对象自己的局部坐标系，用距离来指定的。图 3-6 展示了与局部坐标系 x 、 y 和 z 轴相关的方向。

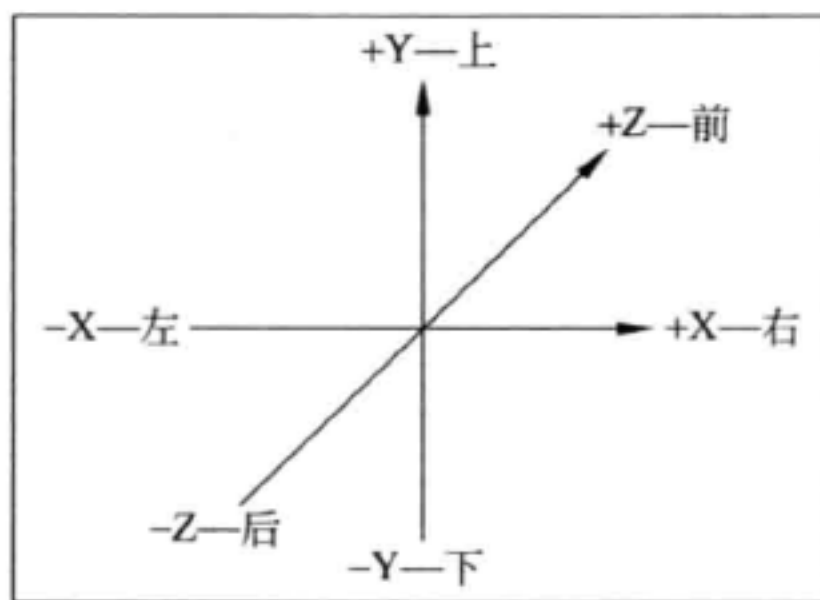


图 3-6 与局部坐标轴相关的方向

下面的这段代码，把原始立方体 3D 对象向前移 10 单位，向右移 20 单位，且向下移 30 单位：

```
var cube:Cube=new Cube();
cube.moveForward(10);
cube.moveRight(20);
cube.moveDown(30);
```

图 3-7 展示了在执行上述代码后，3D 对象每次移动的情形。

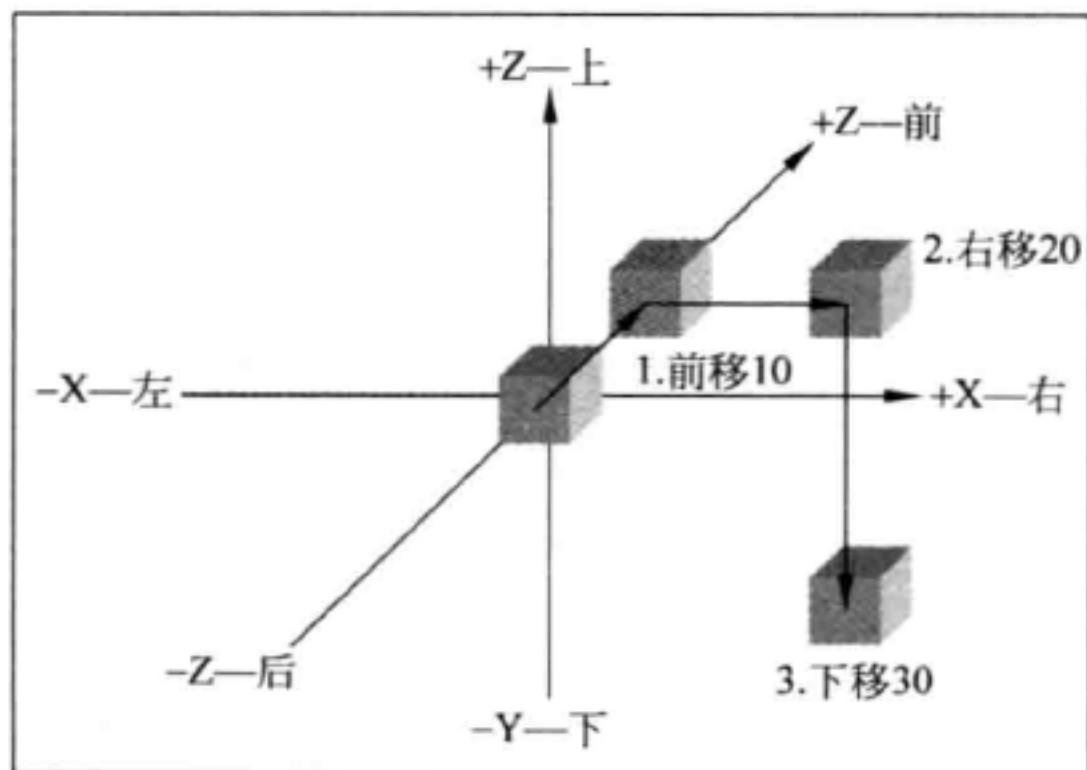


图 3-7 3D 对象的移动

3.3.4 moveTo()函数

移动函数 `moveTo()` 的功能方式与 `position` 属性非常相似,它能够一次指定 3D 对象相对于父空间里沿所有三个坐标轴的位置。

在下面的例子中调用对象的 `moveTo()` 函数,完成与通过把向量 `Vector3D(20, -30, 10)` 赋值给 3D 对象的 `position` 属性,使对象产生移动相同的结果。

```
var cube:Cube=new Cube();
cube.moveTo(20, -30, 10);
```

3.3.5 translate()函数

转换函数用于在局部空间里,沿任意矢量移动 3D 对象。

它的第一个参数轴向矢量 `axis`,定义了移动的方向,当计算移动的距离时并不使用轴向矢量的长度(轴向矢量 `axis` 是经过标准化的,它是单位向量,于是在引用 `translate()` 函数时,轴向矢量有单位长)。第二个参数 `distance` 是距离,它定义了 3D 对象将沿矢量移动多远。

下面的例子将移动立方体到原先用其他移动函数把它移动到的同样位置。我们知道轴向矢量 `axis` 将使立方体移动的方向与在上面的其他例子中立方体移动的方向完全相同,这是因为在构造立方体时,使用了同样的 `x`、`y`、和 `z` 坐标轴的值(即 20、-30 和 10)。只是为了方便起见,构造立方体时才用上述这些轴向矢量 `axis` 值。要记住,因为轴向矢量 `axis` 是经过标准化的,它仅仅是向量指定的方向。如果指明一个向量是 $(2, -3, 1)$ 或 $(2/3, -1, 1/3)$, 则它们的指向完全相同。

这样,我们知道轴向矢量 `axis` 仅定义了移动的方向,但还不知道要移动多远。因此,要将立方体移动到与上面其他例子相同的最终位置,必须知道向量 $(20, -30, 10)$ 的长度,这可以用几何里的勾、股、弦定理计算出来,即向量各坐标轴分量的平方和的平方根:

```
Math.sqrt(20 * 20 + (-30) * (-30) + 10 * 10)(或 Math.sqrt(1400))
```

代码如下:

```
var cube:Cube=new Cube();
cube.translate( new Vector3D(20, -30, 10), Math.sqrt(1400));
```

3.3.6 修改旋转

Away3D 包含 10 个以上的函数和属性以用于旋转一个 3D 对象。

1. 旋转初始化对象参数

一个 3D 对象最初的转动,经常用 rotationX、rotationY 和 rotationZ 初始化对象参数来指定,这些值代表一个 3D 对象围绕在父空间里的 x、y 和 z 轴的转动,并且以度来测量。

Away3D 函数接受的角度经常以度为单位,而 Flash 函数,如 Math.sin、Math.cos 和 Math.tan 全都以弧度为单位。因此要留心,在使用不同的函数时,角度的量度单位是不同的。

把弧度换算成度时可使用公式:

$$\text{度} = \text{弧度} \times 180 / \pi$$

度换算成弧度时可使用公式:

$$\text{弧度} = \text{度} / 180 \times \pi$$

```
var sphere:Sphere=new Sphere(  
    {  
        rotationX:10,  
        rotationY:20,  
        rotationZ:30  
    }  
);
```

2. 旋转参数

3D 对象的转动,在它建立时也能通过它自身的 rotationX、rotationY 和 rotationZ 属性来设置。

```
var sphere:Sphere=new Sphere();  
sphere.rotationX:10;  
sphere.rotationY:20;  
sphere.rotationZ:30;
```

3. rotateTo()函数

旋转函数 rotateTo(),调用这个函数,设置 3D 对象围绕父坐标 x、y 和 z 轴的旋转。下列代码可以达到与原来描述的通过设置 rotationX、rotationY 和 rotationZ 属性完全相同的效果。

```
var sphere:Sphere=new Sphere();  
sphere.rotateTo(10,20,30);
```

4. 欧拉属性

欧拉属性与 `rotateTo()` 函数功能非常相似,只不过用一个向量对象来替换 `rotateTo()` 函数的三个数值。

```
var sphere:Sphere=new Sphere();  
sphere.eulers=new Vector3D(10,20,30);
```

5. rotate() 函数

前面所述的旋转函数,全是围绕父空间的 x 、 y 和 z 轴转动 3D 对象。将这些围绕固定轴的转动联合使用,可以把 3D 对象转动到任意想要的位置。然而在某些情况下,把 3D 对象围绕一个任意轴转动,将是更容易和更快的。旋转函数 `rotate()` 就正好能用于这种情况。

下列代码将旋转球体对象围绕局部空间向量 $(1,0,1)$ 旋转 90° 。

```
var sphere:Sphere=new Sphere();  
sphere.rotate=(new Vector3D(1,0,1),90);
```

6. lookAt() 函数

朝向函数 `lookAt()` 用于指定 3D 对象(通常是照相机)局部坐标的 Z 轴朝向父空间里的一个位置,由第一个参数 `target` 定义,可以根据需要指定 3D 对象的滚动,使 3D 对象转动到面向第二个参数指定的位置,如图 3-8 所示。

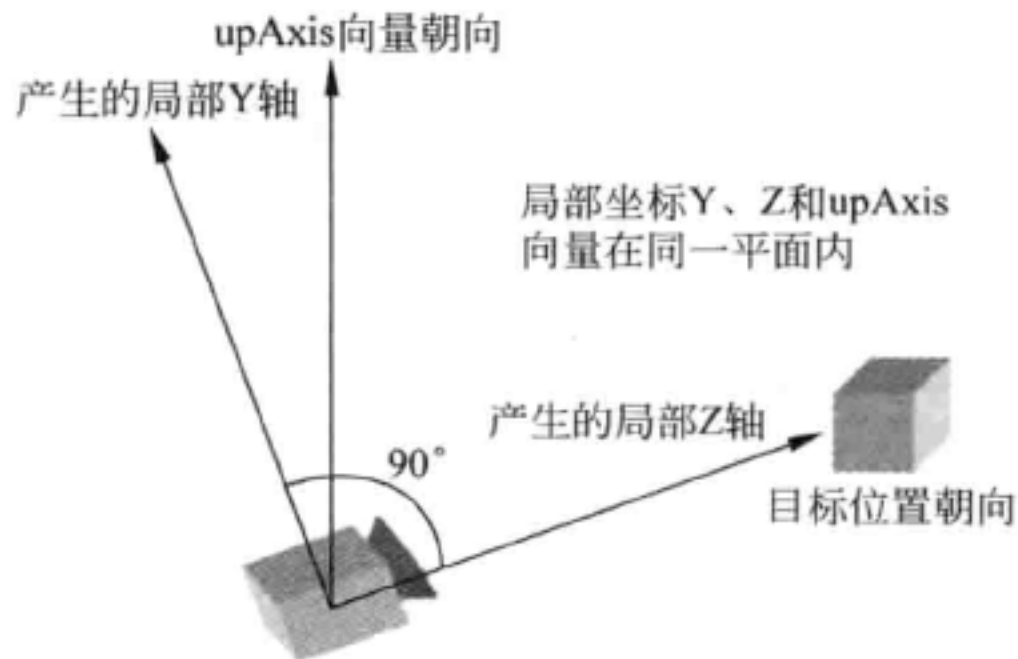


图 3-8 3D 对象的朝向变换

第二个参数是 `upAxis`,这是在父空间里的向量,用于与 3D 对象局部 Z 轴结合到一起旋转到面向定义平面的目标位置,然后 3D 对象的局部 Y 轴位于这个定义平面上,而与局部 Y 轴构成直角边的方向就是局部 Z 轴方向。

如果第二个参数 `upAxis` 没有提供,将使用默认值 $(0,-1,0)$ 。

下列代码将使照相机从它的父容器原点旋转到朝向 $(10,20,30)$ 方位。

```
camera.lookAt(new Vector(10,20,30));
```

7. 枢轴点属性

迄今为止,详述的各默认旋转函数和属性,将使 3D 对象围绕它的父容器的各坐标轴旋转,或围绕它的父空间的向量轴旋转,好似 3D 对象坐落在父容器的原点一般。图 3-9 中的立方体,从上面看围绕 Y 轴旋转了 45° 。

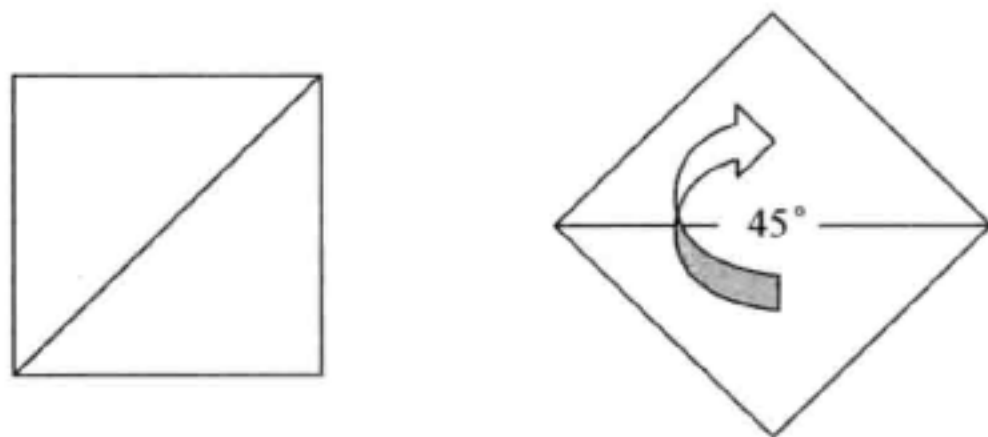


图 3-9 立方体沿 Y 轴旋转 45°

使 3D 对象围绕一外部点旋转,这也是可能的,这样的外部点称为枢轴点,非常像老式的钟摆。通过给定一个 Vector3D 对象的 pivotPoint 属性,定义枢轴点的位置。

定义枢轴点的位置在局部空间里。这里设置枢轴点的位置在立方体右面 200 单位。

```
var cube:Cube=new Cube();
cube.pivotPoint=new Vector(200,0,0);
cube.rotationY=45;
scene.addChild(cube);
```

现在, `cube.rotationY=45` 不是使立方体围绕 Y 轴旋转 45° ,而是围绕右边的枢轴点旋转 45° ,如图 3-10 所示。

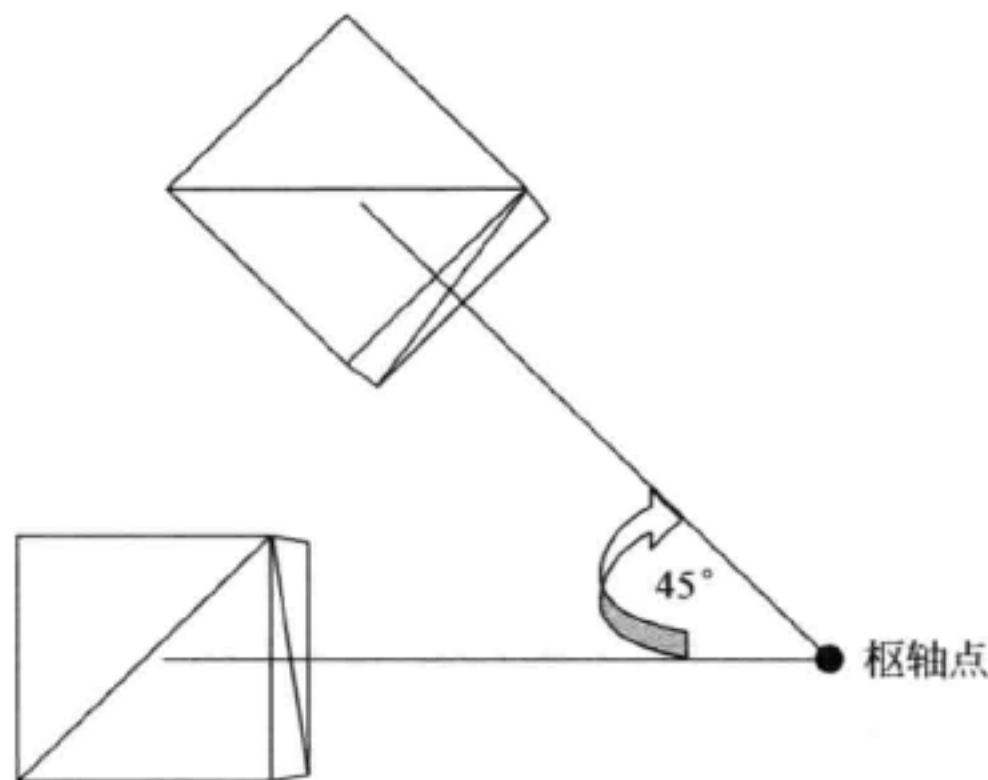


图 3-10 正方体沿枢轴点旋转 45°

8. movePivot() 函数

移动枢轴点 `movePivot()` 函数也能用于设置枢轴点的位置。`pivotProperty` 属性和 `movePivot()` 函数之间的不同点是, 当用 `movePivot()` 函数时, 无须建立一个中间的 `Vector3D` 对象。

下列代码设置 `pivotProperty` 属性在点 $(200, 0, 0)$, 与上面的代码有相同的效果。

```
var cube:Cube=new Cube();
cube.movePivot(200,0,0);
cube.rotationY=45;
scene.addChild(cube);
```

9. 场景枢轴点属性

场景枢轴点 `scenePivotPoint` 属性提供了一个在全局空间里找到枢轴点位置的方法。`scenePivotPoint` 属性是只读的, 因此不能通过它设置枢轴点的新位置。

```
var cube:Cube=new Cube();
cube.movePivot(200,0,0);
var scenePivot:Vector3D=cube.scenePivotPoint;
```

10. pitch()、roll()和 yaw() 函数

倾斜 `pitch()`、滚动 `roll()` 和偏航 `yaw()` 函数可以分别沿着局部空间里的 X 轴、Z 轴和 Y 轴旋转 3D 对象。如果把这些函数用于照相机, 这些概念是很容易想象的。修改 `pitch` 将使照相机朝向上或向下, 修改 `yaw` 将使照相机朝向左或向右, 修改 `roll` 将使照相机滚动朝顺时针或相反, 如图 3-11 所示。

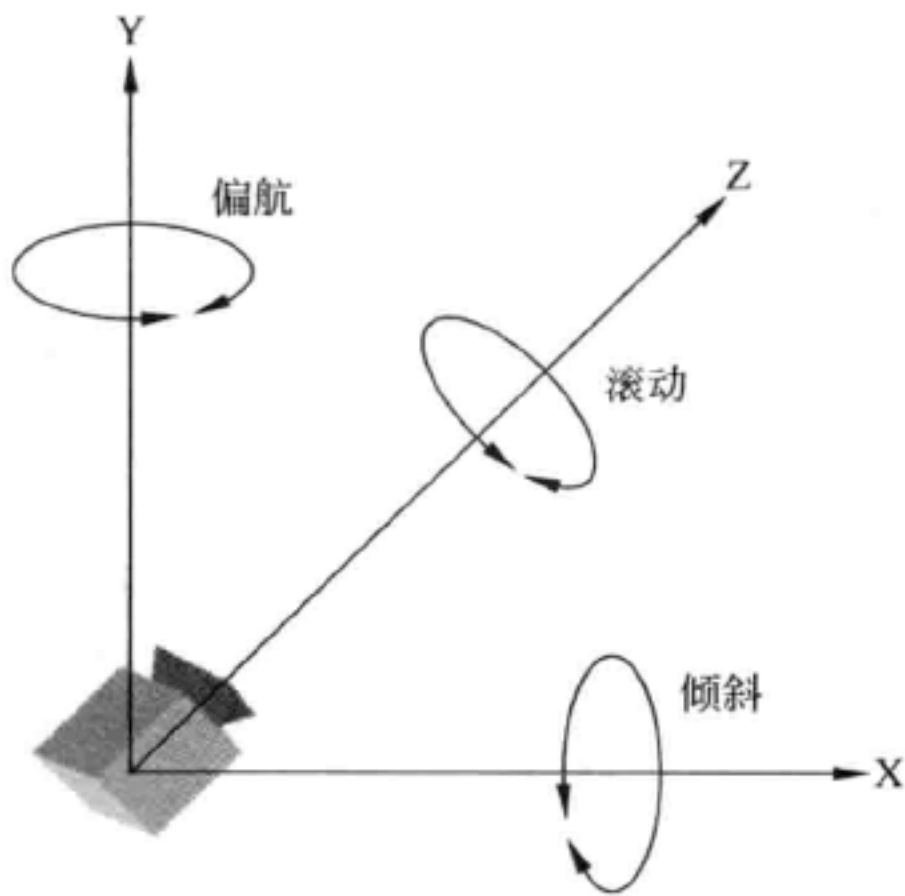


图 3-11 局部空间中各坐标轴方向的旋转函数

```
var sphere:Sphere=new Spere();  
    sphere.pitch(90);  
    sphere.yaw(90);  
    sphere.roll(180);
```

3.3.7 修改缩放

缩放一个 3D 对象,涉及修改它在自己的局部坐标轴上的尺寸,一个 3D 对象能统一地沿所有三个坐标轴缩放或单独地沿局部 X、Y 或 Z 轴缩放。

1. 缩放初始对象

一个 3D 对象沿它自己全部局部坐标轴的初始缩放,经常用 scale 初始对象的参数来指出。缩放 3D 对象统一沿所有三个坐标轴一起缩放。在这个例子中已经将球的默认尺寸放大两倍。

也可以给这些缩放函数和参数提供负的数值,可实现翻转 3D 对象的“反”效果。

```
var sphere:Sphere=new Sphere(  
{  
    scale: 2  
})
```

2. scale() 函数

缩放函数一次应用到所有局部坐标轴上,达到与 scale 初始化对象参数有相同的效果,并能用于那些没有实现初始化对象的 3D 上。

```
var sphere:Sphere=new Sphere();  
sphere.scale(2);
```

3. scaleX, scaleY 和 scaleZ 属性

一个 3D 对象沿它的各坐标轴缩放,通过设置 scaleX、scaleY 和 scaleZ 属性来指明。这里使用这三个属性来缩放一个 3D 球对象,其中 2 是沿 X 轴的放大系数,3 是沿 Y 轴的放大系数,4 是沿 Z 轴的放大系数。

```
var sphere:Sphere=new Sphere();
```

```
sphere. scaleX=2;  
sphere. scaleY=3;  
sphere. scaleZ=4;
```

3.3.8 修改转换

定位、旋转和缩放一个 3D 对象的最终结果能够描述为一个 4×4 的矩阵,叫做转换矩阵。

尽管我们仅工作在三维空间里,仍然需要一个 4×4 矩阵去容纳 Away3D 支持的全部转换信息,在一个矩阵里包括缩放、旋转和转换的信息。

矩阵由在 flash.geom 包里的 Matrix3D 类代表,直接操作转换矩阵也是可以的,然后通过定义在 Object3D 类里的 transform 属性将它传递给 3D 对象。然而使用列出的函数去转换 3D 对象,一般来说这样做是更加方便的,而不是直接修改转换矩阵。

3.3.9 渐变操作

到目前为止所介绍的大量的应用程序中,对 onEnterFrame() 函数里的每一帧场景中的 3D 对象都稍微地做了一些转换。另一个通常用于修改 3D 对象属性包括定义为转换属性在内的方法,是 Tweening 库。有很多的免费库能执行 Tweening 操作,其中之一是 GreenSock 的 TweenLite 库,TweenLite 库能从网站 <http://www.greensock.com/tweenlite> 下载。

虽然它能免费下载,但 TweenLite 仍有许可证限制,可参见 <http://www.greensock.com/licensing/>。

为使用 TweenLite 库,可把它添加到 Flash Builder 和 Flash CS4 的 Source path 或 FlashDevelop 的 Project ClassPaths 里,正如第 1 章中使用 Away3D 库一样。

Tweening 库有很多功能函数,但实现 Tweening 操作,还是相当简单的。为了展示 TweenLite 能用于 3D 对象,新建一个叫做 TweeningDemo 的应用程序。

```
package  
{  
    import flash.geom.Vector3D;  
    import away3d.primitives.Sphere;
```

输入 TweenLite 类,使它在 TweeningDemo 应用程序中是可用的。

```
import com.greensock.TweenLite;
public class TweeningExample extends Away3DTemplate
{
    protected var sphere:Sphere;
    public function TweeningExample()
    {
        super();
    }
    protected override function initScene():void
    {
        super.initScene();
```

照相机位于 Y 轴正方向 2000 单位和 Z 轴负方向 2000 单位,然后调用 lookAt() 函数,改变照相机的朝向,这样照相机就看到了场景的原点。在这样的位置和朝向下,照相使我们从高处以稍为小的视角观察到了 X、Y 平面,这是观察球体在 X、Y 平面上运动的最好位置。

```
camera.position=new Vector3D(0,2000,-2000);
camera.lookAt(new Vector3D(0,0,0);
```

然后建立一个原始球体,并将其添加到场景里。因为没有指定球的结构函数的值,则球体的初始位置在场景的原点。

```
sphere=new Sphere();
scene.addChild(sphere);
```

然后调用 tweenToRandomPosition() 函数完成缓动初始化的第一次操作。

```
tweenToRandomPosition();
}
protected function tweenToRandomPosition():void
{
```

在 tweenToRandomPosition() 函数里,调用了 TweenLite 类定义的静态函数 to(),用于渐进地多次修改一个对象的属性。

第一个参数 target,是一个将要用缓动操作修改的对象。在本例中,对象就是球体。

第二个参数 duration,定义了缓动操作持续的秒数。在这里使用值 1,它将 1s 的周期内,使球体的属性渐进地从当前值修改为新值。

```
tweenLite.to(sphere,1,
{
```

第三个参数,是一个用文字符号创建的对象 vars,这与用 Away3D 创建一个初始化对象的方式相同,在这个对象中,指定了希望 3D 对象完成一次完整的渐进操作的值。这些属

性 `x`, `z`, `scaleX`, `scaleY`, `scaleZ` 和 `rotationY` 都属于球体对象要显示出来的, 在这个例子中, 已经指定这些属性的值都是随机值。

对象 `vars` 的最后一个属性 `onComplete`, 是一个特殊属性, 仅能为 `TweenLite` 类认识, 任何指定给这个属性的函数都要完成一次渐进操作。这里已经指定了 `tweenToRandomPosition()` 函数。由于 `tweenToRandomPosition()` 函数里建立的渐进操作是连续的渐进操作, 当一个渐进操作完成时, 就调用 `tweenToRandomPosition()` 函数, 开始一个新的渐进操作。

使用 `onComplete` 属性排列渐进操作的一个数, 能建立一些非常复杂的运动, 或照脚本操作。

```
x: Math.random() * 1000-500,  
z: Math.random() * 1000-500,  
scaleX: Math.random() * 1.5+0.5,  
scaleY: Math.random() * 1.5+0.5,  
scaleZ: Math.random() * 1.5+0.5,  
rotationY: Math.random() * 180,  
onComplete: tweenToRandomPosition  
}  
};  
}  
}
```

当应用程序运行的时候, 球在场景中沿 `X` 轴和 `Y` 轴 $-500 \sim 500$ 单位内随机地确定一个位置而运动, 球的缩放是它原尺寸的 $0.5 \sim 2$ 倍, 并围绕局部 `Y` 轴 $0^\circ \sim 180^\circ$ 任意转动。

渐进操作的最大好处之一, 是启动渐进操作后就不用管它了, 不必跟踪每帧里传递了多少次渐进操作, 以及手工修改每帧属性的渐进操作次数, 如在原先的程序 `onEnterFrame()` 中所做的那样, 事实上可以看到, 在本例中完全没有使用 `onEnterFrame()` 函数。

`TweenLite` 包含的功能比本书里涉及的功能要多, 来自 `GreenSock` 的 `TweenMax` 库包含更多的功能, 为浏览这些附加的功能, 可以在网站 <http://www.greensock.com> 找到交互式的演示范例, 学习修改屏幕上的 2D 对象的位置, 缩放它的大小和旋转它。只需要记住, `Tweening` 库通常没有 2D 的固定的概念, 随着时代进步的要求, 3D 仅修改了给出的属性的值。2D 对象与 3D 对象间的唯一不同是相对第三维沿 `Z` 轴属性的修改。

3.4 嵌套

当我们看到父坐标系统的时候,把 3D 对象加到父容器而不是加到场景也是可以的。把 3D 对象加到父容器里,这样的方法叫嵌套。嵌套用于同步地转换一组 3D 对象。父容器能够移动、缩放、旋转,转而它的 3D 对象儿子也同步转换。

下面看一个在运动中嵌套的实际例子。设想建立一个射击游戏,里面每一个空间飞船能配置各种枪炮,每一个枪炮由有区别的 3D 对象代表。为了完成这个要求,要提供飞船与枪炮组合的这两组分开的模型。随组合模型数量的增加,这样的方法将很快地变成不切实际的构想。例如,如果有 5 个飞船,每个飞船能够配置 6 种枪炮,那将需要提供 30 种组合模型。

比较好的解决办法是把每个飞船和枪炮的模型分开,在程序运行时,当需要模型组合的时候,再组合它们。

图 3-12 是枪炮的屏幕截图。



图 3-12 枪炮模型截图

图 3-13 是飞船的屏幕截图。



图 3-13 飞船模型截图

图 3-14 是飞船和枪炮的组合屏幕截图。



图 3-14 枪炮和飞船模型组合截图

以下的 NestingDemo 类的代码展示了把显示在屏幕截图中的飞船和 3D 枪炮对象添加到一个容器里,以作为一个组来转换它们。

```
package
{
    import away3d.containers.ObjectContainer3D;
    import away3d.core.base.Mesh;
```

Cast 类提供了一个方便的在不同类型间投射不同对应类的方法。在这个例子中,Cast 类用于把 BitmapMaterial 类与加到场景中 3D 对象上的材质对象相结合,涉及材质更详的信息,请参见第 5 章。

```
import away3d.core.utils.Cast;
import away3d.materials.BitmapMaterial;
import flash.events.Event;
public class NestingDemo extends Away3DTemplate
{
    //嵌入一个纹理 texture.jpg 文件,它通过纹理 texture 类来访问。
    [Embed(source="texture.jpg")]
    protected var Texture:Class;
    //把飞船和枪炮模型加到里面的父容器引用 container 属性。
    protected var container:ObjectContainer3D;
    public function NestingDemo()
    {
        super();
    }
    Protected override function initScene():void
    {
        super.initScene();
```

/* 在初始化场景函数 initScene()中,建立一个新的位图材质 BitmapMaterial 对象,然后将其应用到

```

飞船和枪炮 3D 对象上 */
    var material:BitmapMaterial=new
        BitmapMaterial(Cast.bitmap(Texture));
/* Fighter 类是一个复杂的 3D 对象,已输出到一个 ActionScript 类, Fighter 类与第 2 章中的
SeaTurtle 类相似,由 fighter 变量来引用,代表飞船,把模型输出到 ActionScript 类可参见第 6 章 */
    var fighter:Mesh=new Fighter();
//然后,把材质施加到 3D 对象.
    fighter.material=material;
/* 枪炮 3D 对象的建立与建立飞船的方法非常相似.下面建立一个新的枪炮 3D 类实例,并把材质加
到它的上面 */
    var gun1:Mesh=new Gun();
    gun1.material=material;
/* 在这样通过初始化对象还没有传递到基础网格类的情况下,建立了用于输出枪炮 3D 类的工具.
这就表示 3D 对象的位置不能用初始对象设置.于是在对象建立之后,只能通过 x、y 和 z 属性来设
置 */
    gun1.x=-150;
    gun1.y=75;
    gun1.z=-115;
/* 飞船有两个枪炮,下面建立第二个枪炮 3D 类实例,第二个枪炮 3D 类位于 X 轴的对面 */
    var gun2:Mesh=new Gun();
    gun2.material=material;
    gun1.x=150;
    gun1.y=75;
    gun1.z=-115;
/* 到此为止,共有三个 3D 对象,一个飞船和两个枪炮.我们想要能使这三个 3D 对象的工作像一个
项目一样.这里嵌套是非常有用的.为此,建立一个新的容器,并将这三个 3D 对象作为第一批参数提
供给新的容器 ObjectContainer3D 的结构函数 */
    container=new ObjectContainer3D(
        fighter,gun1,gun2,

```

也可使用下面的代码,在容器建立时,将 3D 对象分别添加到容器
中。 */

```

    container.addChild(fighter);
    container.addChild(gun1);
    container.addChild(gun2);

```

最后的参数是初始化 int object 对象,它将把容器 container 设置在沿 z 轴正方向的 2000 个单位 */

```

{
    Z:2000
}
);
scener.addChild(container);

```



```
    }  
    protected override function onEnterFrame(event:Event):void  
    {  
        super.onEnterFrame(event);  
        /* 通过把飞船和两个枪炮 3D 对象嵌套到一个容器,现在,能把它们当作一个组来转换.事实上,在  
        建立容器的时候,已经移动为一个组,指定给容器的初始位置.通过修改容器的旋转属性,  
        onEnterFrame()函数也把 3D 对象当成一个组来转换 */  
        ++container.rotationX;  
        ++container.rotationY;  
    }  
}
```

当这个程序运行的时候,可以看到,尽管父容器在旋转、移动,而容器中的飞船和两个枪炮 3D 模型,依旧保持彼此的相对位置。

景 深 排 序

正确地排序场景中的 3D 对象,是能否正确地在屏幕上画出场景的关键。Away3D 使用画家算法在屏幕上画出组成场景的元素,在大多数情况下,Away3D 能正确地排序并画出这些元素显示场景。可能有些读者认为这样的过程是很容易的,然而在有些情形下,对在场景中 3D 对象的 Away3D 排序做调整是必需的。本章展示了这样的情形,并介绍了能手工改正排序过程的方法。

Away3D 也包含一些附加的渲染器,它能用于自动地调整 3D 对象在场景中的排序。展示出这些渲染器,并探索了它们可能对 Away3D 应用程序执行性能的影响。

本章主要内容:

- 画家算法
- 如何排序场景
- 如何影响或强制 3D 对象的排序次序
- 一些附加的渲染器

4.1 画家算法

画家算法参考了被很多画家所采用的绘画技法,在这种技法里,首先画出场景里最远的部分,然后画出场景中较近的部分,逐次地画在它前一幅的上面。图 4-1 来自维基上的有关文章,显示出绘画野外景象时所采用的一些步骤。

景象中最远的背景山首先画出,草地和灌木然后画出,最后画出的树林覆盖在它们

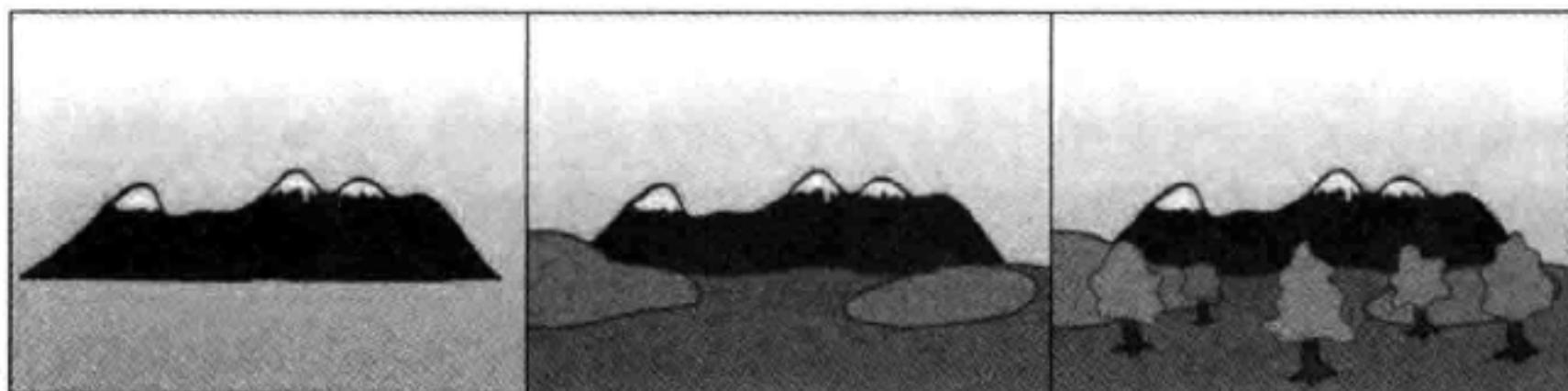


图 4-1 按步骤绘出的野外景象

上面。

Z-排序或景深排序是一种对组成 3D 对象的元素进行排序的技术,这种排序方法是根
据相机观察这些元素时的远近情况而排列的。然后允许这些 3D 对象元素用画家算法,以
从场景中最远的背景开始到最近的背景顺序依次描绘到屏幕上。

在大多数情况下,这个算法是没有问题的,并没有附加的必要步骤以改正后的顺序画出
3D 对象元素。然而在有些情形下,这个算法是失败的,为理解画家算法失败的情形,首先看
看组成一个 3D 对象的元素在场景里是如何排序的。

4.2 场景排序

场景中每一个元素的距离用一个数来表示,叫景深,或叫 z 深度。计算景深,使用组成
该元素的每个顶点沿照相机局部坐标系 Z 轴的位置的平均值表示。有一个很容易的方
式来想象照相机局部空间,即假定照相机坐落在原点,并直接朝 Z 轴的正方向,在图 4-2 中
阐明了这一情形。

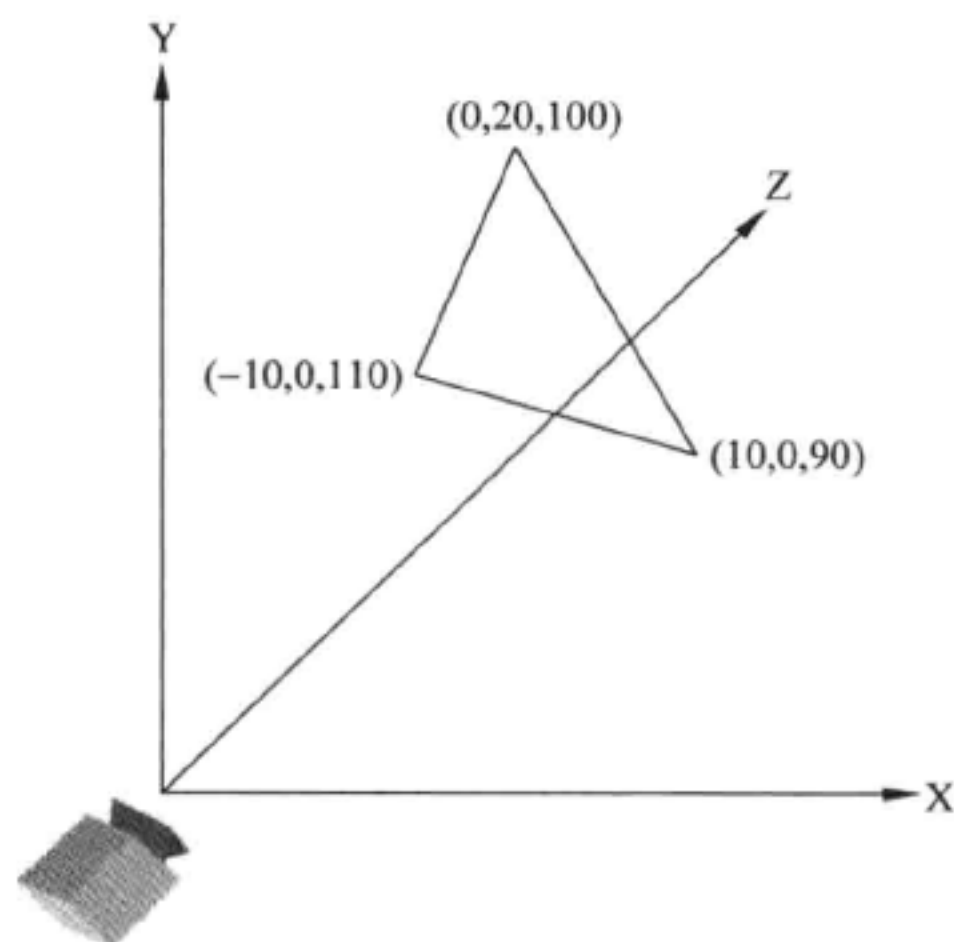


图 4-2 照相机局部坐标系

组成三角形的各顶点的坐标值,标记在图里面,这些坐标值是用于照相机的本地空间里的,为计算三角形的景深—— z 深度,取这些坐标值的 z 分量(即 110,100 和 90),计算平均值取整最后得到 100。

即使各顶点的深度范围是 90~110,以后仍使用值 100 作为此三角形的 z 深度。既然 z 深度的平均值不能精确地描绘场景内原始的相对位置,那么在排序 3D 对象元素时,根据它们的 z 深度就会导致与实际不一致的情形。

为证实 3D 对象元素不能依景深正确地排序这一情况,新建一个叫做 ZSorting 的例子。在初始化场景函数 `initScene()` 中,将建立两个三角形,这两个三角形从照相机的视角来看,一个重叠了另一个。

```
package
{
    Import away3d.primitives.Triangle;
    public class ZSorting extends Away3DTemplate
    {
        public function ZSorting()
        {
            super();
        }
        protected override function initScene():void
        {
            super.initScene();
            camera.z=0;
            var triangleA:Triangle=new Triangle(
            {
                x: -30,
                y: 0,
                z: 500,
                rotationY: -5,
                yUp: false,
                bothsides: true
            }
            );
            var triangleB:Triangle=new Triangle(
            {
                x: 30,
                y: 0,
                z: 499,
                rotationY: 60,
                yUp: false,
                bothsides: true
            }
            );
        }
    }
}
```



```

scene.addChild(triangleA);
scene.addChild(triangleB);
}
}
}

```

图 4-3 是从上往下看的场景样子,三角形 B 景深——z 深度稍小于三角形 A,并且它们之间并不相交。

从上往下看场景时,显然,右边的三角形(Triangle B)应该出现在左边的三角形(Triangle A)的后面,但是由于在计算时,三角形 B 的景深比三角形 A 的景深略小,所以应用程序认为三角形 B 是在三角形 A 的前面,这就导致三角形 B 最后被画出,覆盖在三角形 A 的上面,出现错误,如图 4-4 所示。

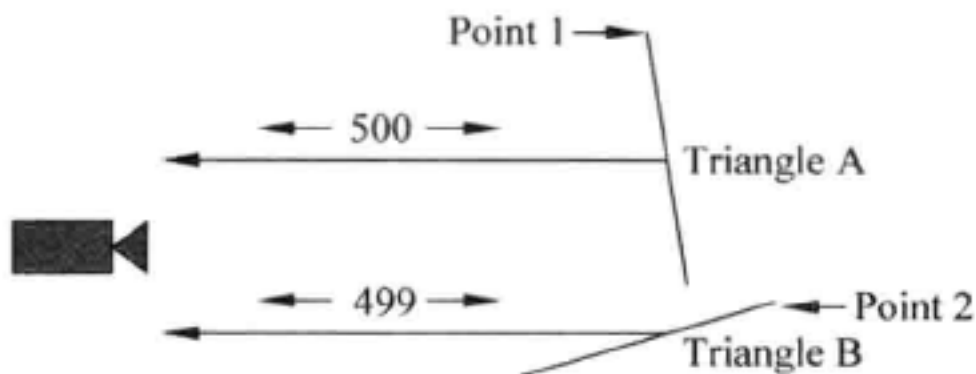


图 4-3 从上往下看场景 3D 对象的景深对比

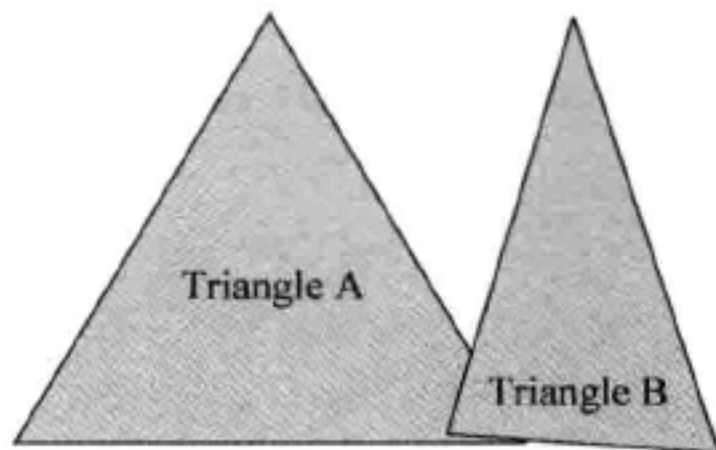


图 4-4 程序运行效果

这里提出的这个例子,证明单一的景深(z 深度)的平均值并不能精确反映实际场景中 3D 对象的景深情况。

4.3 调整排序的顺序

Away3D 中包含很多用于调整排序次序的方法,以这种调整后的排序次序画出原来的元素。在上面提供的例子中画出的排序次序可以修改,即可把三角形 A 带到场景的前面,或者强制把三角形 B 带到场景的后面。

4.3.1 前推和后推属性

前推属性 pushfront 强制画出排序的原来元素,以最靠近照相机的点为根据。对于图 4-3 中,三角形 A 最靠近照相机的点是 Point 1。因为 Point 1 靠近照相机,比三角形 B 的景深更近一些,所以设置三角形 A 的前推属性 pushfront 为 True,这将把三角形 A 带到场景的前面,表示它将被最后描绘出来(在程序中要加入 import away3d.core.base.Object3D; 语句)。

```

var triangleA:Object3D=new Triangle(
{

```

```
        x: -30,  
        y: 0,  
        z: 500,  
        rotationY: -5,  
        yUp: false,  
        bothsides: true,  
        pushfront: true  
    }  
);
```

后推属性 `pushback` 强制画出排序的原始元素,以离照相机最远的点为根据。对于图 4-3 中,三角形 B 离照相机最远的点是 Point 2。因为 Point 2 离相机远,比三角形 A 的景深远些,所以设置三角形 B 的后推属性 `pushback` 为 `True`,这将把三角形 B 带到场景的后面,表示它将被最先描绘出来。

```
var triangleB:Object3D=new Triangle(  
    {  
        x: 30,  
        y: 0,  
        z: 499,  
        rotationY: 60,  
        yUp: false,  
        bothsides: true,  
        pushback: true  
    }  
);
```

4.3.2 屏幕 Z 位移属性

屏幕 Z 位移 `ScreenZOffset` 属性用于强制使一个三角形比其他的三角形离相机远一些,Away3D 将把屏幕 Z 位移 `ScreenZOffset` 属性的值加到 `z` 深度,用这种方法可调整 3D 对象在场景中的相对 `z` 深度。

正的 `ScreenZOffset` 值增加 `z` 深度,强制使 3D 对象更朝向场景的背后。负的 `ScreenZOffset` 值减少 `z` 深度,强制使 3D 对象更朝向场景的前面。请注意:设置 `ScreenZOffset` 值,并不改变 3D 对象在场景中的位置和实际景深,仅改变它们画出的次序。

为了强制把三角形 A 画在三角形 B 上,可以给它指定一个负值的 `ScreenZOffset` 参数。在下面的例子代码中,三角形 A 将有 490 的 `z` 深度,这就把它画到了三角形 B 的前面,而三角形 B 的 `z` 深度是 499。

```
var triangleA:Object3D=new Triangle(  
    {  
        x: -30,  
        y: 0,  
        z: 490,  
        rotationY: 60,  
        yUp: false,  
        bothsides: true,  
        pushback: true  
    }  
);
```

```
        rotationY: -5,  
        yUp: false,  
        bothsides: true,  
        screenZOffset: -10  
    }  
);
```

为了强制把三角形 B 画在三角形 A 的后面, 给它指定一个正值的 ScreenZOffset 参数。在下面的例子代码中, 三角形 B 将有 509 的 z 深度, 这就把它画到了三角形 A 的背后, 而三角形 A 的 z 深度是 500。

```
var triangleB:Object3D=new Triangle(  
    {  
        x: 30,  
        y: 0,  
        x: 499,  
        rotationY: 60,  
        yUp: false,  
        bothsides: true,  
        screenZOffset: 10  
    }  
);
```

Away3D 文档叙述, screenZOffset 的值仅对于当 3D 对象的 ownCanvas 的属性设置为 true 时有效。而在 Away3D 3.6 里设置 screenZOffset 是不正确的, 无论 ownCanvas 的属性设置为 true 或 false, 设置 screenZOffset 的属性均不被认可。

4.3.3 画布属性

强制 3D 对象排序的最后一种方式, 是把它们画在自己的画布上。为此, 就要设置 ownCanvas 属性为 true, 并强制设置 canvas 的 z 深度。

一个 canvas 是一个层, 3D 对象就画在它上面。它们的工作方式与其他图像编辑软件 Photoshop 里层的概念非常相似。3D 对象能画到画布里, 然后画布画在场景中任何其他 3D 对象的旁边。

首先, 设置 ownCanvas 的属性为 true。

```
var triangleB:Object3D=new Triangle(  
    {  
        x: 30,  
        y: 0,
```

```

    x: 499,
    rotationY: 60,
    yUp: false,
    bothsides: true,
    ownCanvas: true
  }
);

```

然后通过设置 `ownSession.screenZ` 的属性来指定 `canvas` 的屏幕深度。因为三角形 A 离照相机 500 单位,所以指定用来显示三角形 B 的画布 `canvas` 的 `z` 深度,稍大一点如 510,这就表示三角形 A 离照相机近一些。

```
triangleB.ownSession.screenZ=510;
```

在图 4-5 中显示了画布的位置,画有三角形 B 的画布的 `screenZ` 值比三角形 A 的 `z` 深度大,我们已经强制把画有三角形 B 的画布作为三角形 A 的背景。

请注意,当通过 `screenZ` 的属性,设置画布 `canvas` 屏幕深度的时候,不像其他修改 3D 对象 `z` 深度的方法,`screenZ` 的属性是一个绝对的值,并不把照相机的相对位置考虑进去。如果照相机位于 $(0,0,-1000)$ 的位置且这是默认值,那么,前述的代码将把画有三角形 B 的画布 `canvas` 画到三角形 A 的前面,这是因为三角形 A 相对照相机的 `z` 深度是 1500,而画布 `canvas` 的 `z` 深度设置为绝对值 510 的原因。

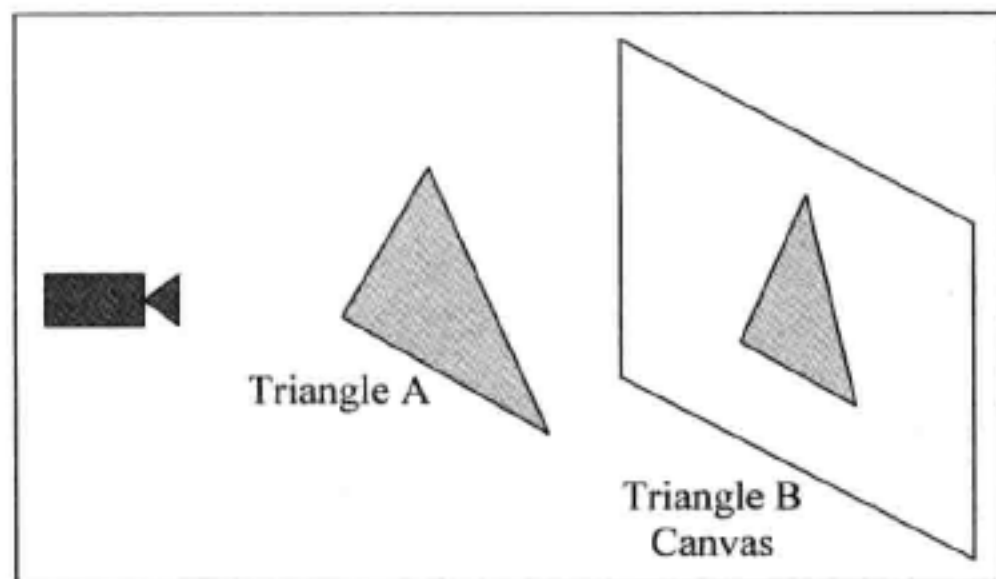


图 4-5 画布对象景深与对象景深关系

如果没有指定 `screenZ` 的值,各画布在场景中的排序是根据各个 3D 对象的 `z` 深度来决定的。这表示在场景中画布的排序,能够通过画在它上面的 3D 对象的 `pushfront`、`pushback` 和 `screenZOffset` 的属性值设置来修改。

4.4 有关 Z-排序的说明

以上所述的所有这些方法,都是通过修改一个 3D 对象相对于场景中其他 3D 对象的景深(z 深度)来实现的。有一点是非常重要的,就是要依据照相机的位置,实现将这些 3D 对象更改到所希望的相对景深(z 深度)。考虑由上面 ZSorting 应用程序建立的同样的场景,现在从与原来相反的位置来看,如图 4-6 所示。

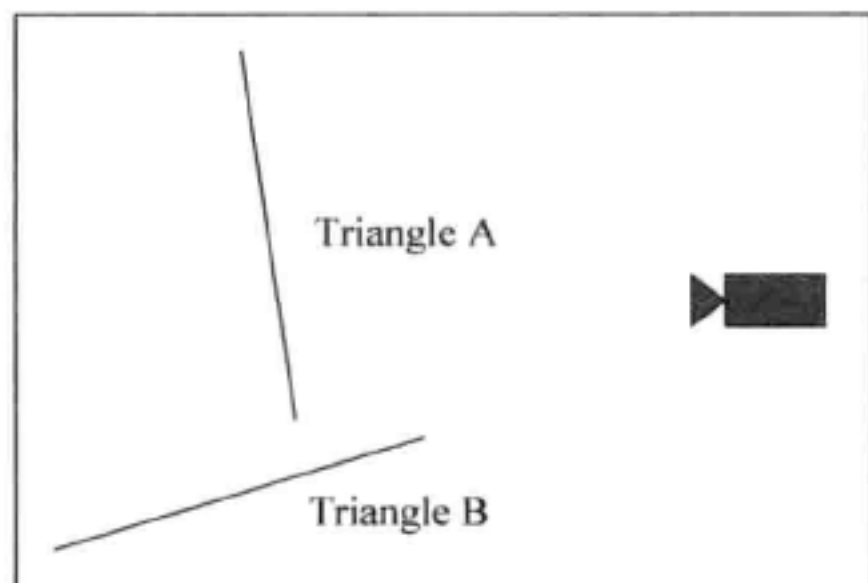


图 4-6 从下往上看场景 3D 对象的景深对比

在这样的情形下,如果设置三角形 B 的 `pushback` 属性为 `true`,同照相机在左边时修改它的绘画次序一样。这在实际中,将会引起一个不可修改 z -排序的错误。这是因为从照相机的现在位置来看,三角形 B 应该画在三角形 A 的前面,而不是在它后面。

于是记住,在指定 `pushback`、`pushfront`、`screenZOffset` 和 `screenZ` 的属性值的时候,可能必须修改场景中相应于照相机位置已经发生改变了的 3D 对象的位置和朝向。

当一个 3D 对象相对于照相机的位置已经改变了的时候,必须修改它的 `screenZOffset` 属性值,请参见第 10 章。

4.5 附加的渲染器

本书中介绍的所有的应用程序,全都使用默认的渲染器。Away3D 包含三种不同类型的渲染器,每个都是通过 `away3d.core.render` 包里的 `Render` 类中的 `static` 属性返回的。

默认的渲染器是用 `BASIC` 属性建立的,其他两个渲染器是用 `CORRECT_Z_ORDER` 和 `INTERSECTING_OBJECTS` 属性建立的,这两个都是四叉树数据结构的渲染器,它们比用默认的 `BASIC` 属性建立的渲染器更加先进。

CORRECT_Z_ORDER 渲染器对 3D 对象在场景中的排序,使用了更加复杂的算法,能解决一些(不是全部)默认渲染器在 z-排序时出现的问题。由 INTERSECTING_OBJECTS 属性返回的渲染器,更深一步使用拆分交叉三角形的方法。

使用这些渲染器是相当简单的,首先引入渲染器 Renderer 类。

```
import away3d.core.render.Renderer;
```

然后,在初始化函数 initEngine() 中,把从渲染器 CORRECT_Z_ORDER 或 INTERSECTING_OBJECTS 属性返回的对象传递给 View3D 对象的结构函数。

```
protected function initEngine():void
{
    scene=new Scene3D();
    camera=new Camera3D();
    view=new View3D(
    {
        renderer: Renderer.INTERSECTING_OBJECTS
    }
    );
    scene=view.scene;
    camera=view.camera;
    addChild(view);
    view.x=stage.stageWidth/2;
    view.y=stage.stage.Height/2;
}
```

通过修改 View3D 的 renderer 属性,也能改变在运行中有效的渲染器。

```
view.renderer=Renderer.CORRECT_Z_ORDER;
```

RenderersDemo 应用程序,允许在程序运行的时候,通过按 1,2 和 3 数字键,在三种渲染器间切换,这提供了一个比较三种渲染器是如何处理交叉 3D 对象的方法。

```
package
{
    import away3d.cameras.Camamera3D;
    import away3d.containers.Scene3D;
    import away3d.containers.View3D;
    import away3d.core.render.Renderer;
    import away3d.materials.WireColorMaterial;
    import away3d.primitives.Cube;
    import away3d.primitives.Triangle;

    import flash.display.Sprite;
```

```
import flash.events.Event;
import flash.text.TextField;
public class RenderersDemo extends Away3DTemplate
{
    private var triangle;
    private var box:Cube;
    public function RenderersDemo():void
    {
        super();
    }
    protected override function initScene():void
    {
        super.initScene();
        triangle=new Triangle(
        {
            edge: 150,
            bothsides: true,
            yUp: false,
            material: new WireColorMaterial(0x224488),
            z: 500
        });
        scene.addChild(triangle);
        box=new Cube(
        {
            z: 500,
            width: 50,
            height: 75,
            depth: 50,
            material: new WireColorMaterial(0x228844)
        });
        scene.addChild(box);
    }
    protected override function initListeners():void
    {
        super.initListeners(); stage.addEventListener(KeyboardEvent.KEY_UP,onKeyUp);
    }
    protected override function initUI():void
    {
        super.initUI();
        var text:TextField=new TextField();
        text.text="Press 1 to enable the BASIC renderer. \n"+
        " Press 2 to enable the CORRECT_Z_ORDER renderer. \n"+
```

```

" Press 3 to enable the INTERRECTING_OBJECT renderer. \n"+
" Press 4 to make the box transparent. \n"+
"Press 5 to make the box opaque. ";
text.x=10;
text.y=10;
text.width=300;
this.addChild(text);
}
protected override function onEnterFrame(event:Event):void
{
    super.onEnterFrame(event);
    triangle.rotationY += 3;
    box.rotationY -= 1;
}
protected function onKeyUp(event:KeyboardEvent):void
{
    switch (event.KeyCode)
    {
        case 49:
            view.renderer=Renderer.BASIC;
            break;
        case 50:
            view.renderer=Renderer.CORRECT_Z_ORDER;
            break;
        case 51:
            view.renderer=Renderer.INTERSECTING_OBJECTS;
            break;
        case 52:
            (box.material as WireColorMaterial).alpha=0.5;
            break;
        case 53:
            (box.material as WireColorMaterial).alpha=1;
            break;
    }
}
}
}

```

从图 4-7 中可看到 INTERRECTING_OBJECTS 渲染器和 BASIC 渲染器是不同的,左边的是默认的 BASIC 渲染器,尽管这两个 3D 对象实际上是交叉的,但三角形显示在立方体的后面。右边的是 INTERRECTING_OBJECTS 渲染器,它把各个三角形表面劈开,用正确的顺序描绘场景。

由以上 RenderersDemo 应用程序建立的场景是简单的,仅包含一个立方体原始模型和一个三角形原始模型。在这样的情形下,在三种渲染器之间的切换,也许在程序运行时没有很明显的效果。但是在一个更加复杂的场景中,将会是怎样的呢?

以下 RenderersPerformanceDemo 应用程序,建立了很多在无形的盒子里到处弹跳的

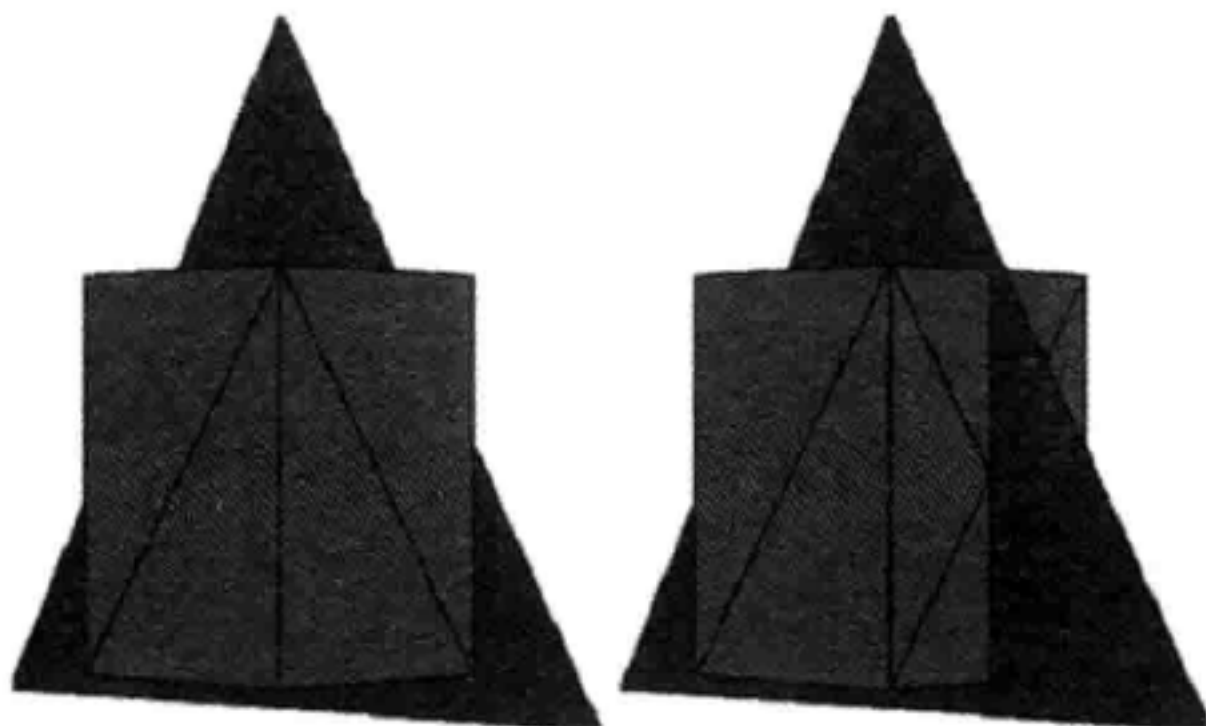


图 4-7 BASIC 渲染器和 INTERRECTING_OBJECTS 渲染器对比图

球体。正像 RenderersDemo 应用程序一样,可在程序运行时,在三种渲染器之间进行切换。

```
package
{
    import away3d.core.render.Renderer;
    import away3d.primitives.Sphere;
    import flash.geom.Vector3D;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
    import flash.text.TextField;
    public class RenderersPerformanceDemo extends Away3DTemplate
    {
        protected static const SPEED: Number=0.5;
        protected static const BOXSIZE: Number=20;
        protected static const NUMBERSPHERES: Number=10;
        protected var spheres:Vector.<Sphere>=new Vector.<Sphere>();
        protected var directions:Vector.<Vector3D>=new Vector.<Vector3D>();
        public function RenderersPerformanceDemo()
        {
            super();
        }
        protected override function initScene():void
        {
            super.initScene();
            this.camera.z=50;
            for (var i:int=0; i< NUMBERSPHERES; ++i)
            {
                var sphere:Sphere=new Sphere(
                {
                    x: Math.random() * BOXSIZE-BOXSIZE/2,
                    y: Math.random() * BOXSIZE-BOXSIZE/2,
                    z: Math.random() * BOXSIZE-BOXSIZE/2,
                    radius: 2
```

```

    }
    );
    spheres.push(sphere);
    this.scene.addChild(sphere);
    directions.push(new Vector3D(
    Math.random()-0.5,
    Math.random()-0.5,
    Math.random()-0.5)
    );
    directions[i] = normalize();
    directions[i] = ScaleBy(SPEED);
    }
    }

    protected override function initListeners():void
    {
        super.initListeners(); stage.addEventListener(Event.ENTER_FRAME, onEnterFrame);
        stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
    }

    protected override function initUI():void
    {
        super.initUI();
        var text:TextField = new TextField();
        text.text = "Press 1 to enable the BASIC renderer. \n" +
        " Press 2 to enable the CORRECT_Z_ORDER renderer. \n" +
        " Press 3 to enable the INTERRECTING_OBJECTS renderer. \n";
        text.x = 10;
        text.y = 10;
        text.width = 300;
        this.addChild(text);
    }

    protected function onKeyUp(event:KeyboardEvent):void
    {
        switch (event.KeyCode)
        {
            case 49:
                view.renderer = Renderer.BASIC;
                break;
            case 50:
                view.renderer = Renderer.CORRECT_Z_ORDER;
                break;
            case 51:
                view.renderer = Renderer.INTERSECTING_OBJECTS;
                break;
        }
    }

    protected override function onEnterFrame(evevt:Event):void
    {
        super.onEnterFrame(evevt);
        for (var i:int = 0; i < 10; ++i)
    }

```

```

{
    var newPosition:Vector3D = new Vector3D();
    newPosition=spheres[i].position.add(directions[i]);
    for each (var property:String in["x","y","z"])
    {
        If (newPosition[property] < -BOXSIZE/2)
        {
            newPosition[property] = -BOXSIZE/2;
            directions[i][property] *=-1;
        }
        If (newPosition[property] > BOXSIZE/2)
        {
            newPosition[property] = BOXSIZE/2;
            directions[i][property] *=-1;
        }
    }
    spheres[i].position=newPosition;
}
}
}

```

以大多数标准衡量,由 10 个球组成的场景,当然是相当简单的,但是要注意:当在从默认的 BASIC 渲染器切换到 CORRECT_Z_ORDER 和 INTERSECTING_OBJECTS 渲染器时,应用程序从平滑的、流畅的变为忽动忽停的,如果不是这样,则会抛出一个 script 超时的错误,如图 4-8 所示。

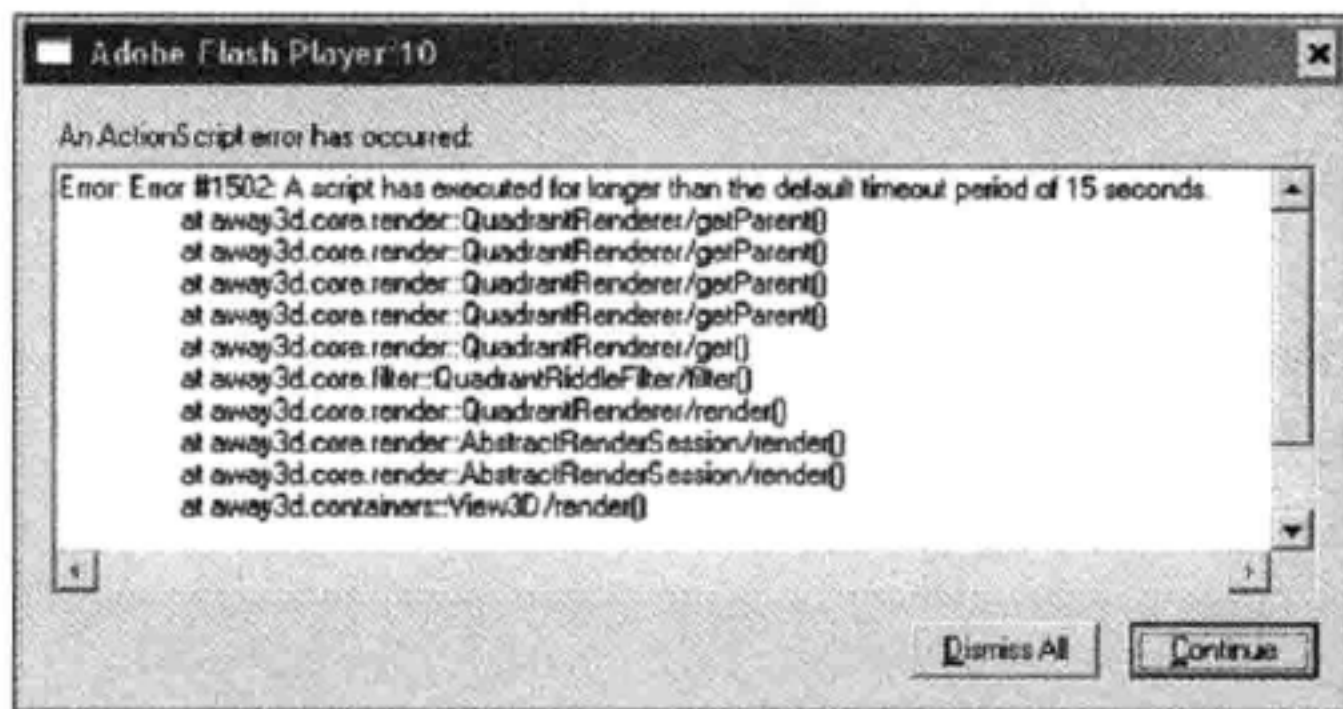


图 4-8 程序运行截图

RenderersPerformanceDemo 应用程序展示了使用更加高级渲染器运行性能的局限性,但对于最简单的场景来说,默认的渲染器还是最好的,它可手工修改 3D 对象排序的次序,以便维护合理的帧速度。

材 质

前面学习了如何在场景中建立显示 3D 对象,并且到现在为止,3D 对象在显示时,使用的都是默认的材质 WireColorMaterial,它显示一个单色的带黑色轮廓的 3D 对象,这样虽然人们能看见 3D 对象,但难免觉得有点单调乏味。实际上,Away3D 包含 10 种以上的材质类型,它们能使 3D 对象显示种类繁多的效果,有些材质使用了新的 Pixel Bender 技术,可显示出原先在 Flash 应用中不能看见的细部的清晰度。

本章主要内容:

- 通过引入或嵌入外部文件,管理资源
- 定义彩色
- Pixel Bender 技术
- Away3D 材质使用的各种着色技术
- 在 Away3D 中能建立的各种材质
- 用光源照明材质

5.1 纹理和材质的区别

本章大量引用了材质和纹理这两个词。一个纹理是一幅图像,就像在使用图像编辑工具软件 Photoshop 或网页制作三剑客里建立的图像一样,然后,纹理被当作材质使用。而材质是一些在 Away3D 里能应用到 3D 对象表面的类。

5.2 资源管理

Away3D 中包含相当多的材质,它们依靠外部的如 png、jpeg 和 gif 这样的图像文件作为它的纹理。处理这些外部图像文件的资源有两种方式:嵌入它们或在运行时访问它们。

ActionScript 包含一个 Embed 关键字,用它能把外部文件直接嵌入到编译的 SWF 文件里。嵌入资源有很多好处:

- (1) 使 Flash 应用程序能发布成一个文件。
- (2) 在运行的时候,访问这些资源不需要等待的时间。
- (3) 可以安全发布,避免与远程访问资源有关联。
- (4) 一旦 SWF 文件下载完成,就没有附加的网络流量。
- (5) SWF 文件能在离线状态运行。
- (6) 嵌入文件能附加地被压缩。

嵌入资源的不利方面,是加大了最后形成的 SWF 文件的大小,导致初始下载时间要长一些。

另一种选择是,这些外部文件可分开来保存,运行时才访问它们,它有以下优点。

- (1) SWF 文件比较小,初始下载时间短。
- (2) 资源文件仅当它们需要时才下载,为以后方便访问,缓冲存储它们。
- (3) 资源文件能够升级,修改而不必重新编译 SWF 文件。

在运行的时候访问分开保存资源文件,也有一些不利的方面。

(1) 在这些资源文件被访问之前,保存这些资源文件的服务器可能必须要配置一下,使资源文件有被访问的权限。

(2) 发布最终的 Flash 应用程序更困难一些,这是由于增加了各个发布文件的数量。

(3) 当把应用程序作为远程资源下载的时候,程序运行将会延迟。

Away3D 支持使用嵌入文件的方法和使用外部资源方法,这两种方法都将在下面分别进行讲述。

当管理资源时,嵌入式资源通常是最好的选项,它防止了很多可能发生的错误,如网络不可靠、安全限制以及很多简单的发布和发表。

然而,对于有些应用程序,当不可能事先预知需要哪些资源的时候,如3D 图像画廊,在这种情况下,装入外部资源是它的唯一选项。也有可能应用程序要装入数据量很大的外部资源,而且这些外部资源又不必立即下载,如大型游戏中玩家不必每次都看到自己或他人的级别。

5.3 在 Away3D 中定义彩色

很多材质的表现外貌,能够通过提供一种颜色来修改。一个最好的例子,就是 WireColorMaterial 材质(当不指定材质时,3D 对象用的就是这一个),它的填充色彩和轮廓线色彩通过 color 和 wireColor 初始化对象参数指定。

在 Away3D 中色彩能够用很多不同的格式定义。所有这些格式的共同特点是:颜色由红、绿和蓝元素色组成,例如,紫色是由红色和蓝色组成的,而黄色是由红色和绿色组成的,等等。

5.3.1 用整数定义色彩

色彩能够定义为一个整数,初始值 int 通常用十六进制数的形式表示,如 0x12CD56,组成初始值 int 的字符,能够是十进制 0~9 之间的数和字符 A~F 之间的字符,可以认为 A~F 之间的字符代表 10~15,则其中的每个字符代表 16 个不同的值。对每个颜色元素而言,00 是最低值,FF 是最高值,前两个字符定义了红色元素,中间两个字符定义了绿色元素,最后两个字符定义了蓝色元素。

有时候必须定义颜色的透明度,这可以通过添加两个附加的字符到十六进制高位数的前面来完成,如 0xFF12CD56。在这里两个领头的字符 FF 定义了颜色的透明度,或颜色的 Alpha 值;然后的 12CD56,分别表示红、绿、蓝三元色。较小的 Alpha 值使颜色更加透明,而较高的 Alpha 值使颜色更加不透明。

5.3.2 用颜色名字的字符串定义色彩

使用整数十六进制的颜色格式,同样也可作为一个串代表,唯一的不同之处就是把前缀 0x 去掉。把 0xFF12CD56 作为串来代表则是“12CD56”或“FF12CD56”。在 MaterialsDemo 类的 applyWireFrameMaterial() 函数里,展示了使用这颜色的格式。

Away3D 也认可很多彩色的名字,图 5-1 中列出了这些色彩的名字,MaterialsDemo 类的成员函数 applyWireFrameMaterial() 也展示了如何用名字定义使用色彩。

random	steelblue	royalblue	cornflowerblue
lightsteelblue	mediumslateblue	slateblue	darkslateblue
midnightblue	navy	darkblue	medinmblue
blue	dodgerblue	deepskyblue	lightskyblue
skyblue	lightblue	powderblue	azure
lightcyan	paleturquoise	mediumturquoise	lightseagreen
darkcyan	teal	cadetblue	darkturquoise
aqua	cyan	truquoise	aquamarine
mediumaquamarine	darkseagreen	mediumseagreen	seagreen
darkgreen	green	forestgreen	limegreen
lime	chartreuse	lawngreen	greenyellow
yellowgreen	palegreen	lightgreen	springgreen
mediumspringgreen	darkolivegreen	olivedrab	olive
darkkhaki	darkgoldenrod	goldenrod	gold
yellow	khaki	palegoldenrod	blanchedalmond
moccasin	wheat	navajowhite	burlywood
tan	rosybrown	sienna	saddlebrown
chocolate	peru	sandybrown	darkred
maroon	brown	firebrick	indianred
lightcoral	salmon	darksalmon	lightsalmon
coral	tomato	darkorange	orange
orangered	crimson	red	deeppink
fuchsia	magenta	hotpink	lightpink
pink	palevioletred	mediumvioletred	purple
darkmagenta	mediumpurple	blueviolet	indigo
darkviolet	darkorchid	mediumorchid	orchid
violet	plum	thistle	lavender
ghostwhite	aliceblue	mintcream	honeydew
lightgoldenrodyellow	lemonchiffon	cornsilk	lightyellow
ivory	floralwhite	linen	oldlace
antiquewhite	bisque	peachpuff	papayawhip
beige	seashell	lavenderblush	mistyrose
snow	white	whitesmoke	gainsboro
lightgrey	silver	darkgrey	grey
lightslategrey	slategrey	dimgrey	darkslategrey
	transparent		

图 5-1 色彩名字

5.4 Pixel Bender

Pixel Bender 对 Flash Player 10 而言是新技术,它实现了用 Pixel Bender 语言编写通用的图像处理技术,用 Pixel Bender 编写的程序被称为 Kernels 或 Shaders,Shaders 的优势

是能跨多个 CPU 和 CPU 内核运行,不像图形处理那样只能通过 Flash 应用程序接口,这给 Shaders 提供了更加快速的处理潜能。

对 Pixel Bender 来说,术语 Kernels 和 Shaders 可以交换使用。

使用 Away3D 3. x 版本胜过使用 Away3D 2. x 版本的优点是它能够使用 Pixel Bender Shader 的 Shaders,当材质类利用 Shaders 时,这些实现过程被隐藏了。这就表示,使用 Shaders 非常像使用正式的材质,而此时,Shaders 提供了一个高清晰度的级别。

通常的错误想法是,Flash Player 10 使用图像处理单元(Graphics Processing Unit, GPU)执行 Shaders,这是不对的。Flash Player 10 不像某些 Adobe 的其他产品使用 Shaders 的方法,当执行 Shaders 时,Flash Player 10 不利用 GPU。

Adobe 已经表明,在将来的 Flash Player 版本中将支持 Pixel Bender 使用图像处理单元。

5.5 光源和材质

光源和材质是 Away3D 中同一创意的两个方面,照明的效果仅能在材质上看到,被照亮的材质将会完整地显示在没有光源的黑色背景上。

Away3D 里包含三种光源类,它们全都来自 Away3D.lights 包,其中的每一个代表了不同的光源。

(1) 点光源。由 PointLight3D 类代表,在场景里的一个点上向所有的方向发射光线,点光源的光强度按平方反比衰减规律计算(光强度=1/距离平方)。

(2) 定向光源。由 DirectionalLight3D 类代表,它发射沿着向量的光线,就像一束探照灯光。不像点光源,定向光源的光强度并不随距离变小,强度的减小随定向光源照亮的矢量与照亮表面间的夹角增大而减小。

(3) 环境光源。由 AmbientLight3D 类代表,同样的光照明全部表面,用于把最小的光量添加到实施材质上。

本文仅阐述了 Away3D 材质的一个子集,并且这些材质可能仅支持不同光源类型中的一个子集。表 5-1 列出了这些能够照亮的材质,以及它们能支持的光源类型和这些材质能否被多种光源照明。

表 5-1 材质及其支持的光源

材质	环境光源	定向光源	点光源	多种光源
Dot3BitmapMaterial	*	*		*
Dot3BitmapMaterialF10		*		
PhongBitmapMaterial	*	*		*
PhongColorMaterial	*	*		
PhongMovieMaterial	*	*		*
PhongPBMaterial			*	
PhongMultipassMaterial		*	*	*
ShadingColorMaterial		*	*	*
WhiteShadingBitmapMaterial		*	*	*

在表 5-1 中,并没有列出确定哪些材质支持哪些光源类型的全部设计。着色法是最好的例子, **PhongMultipassMaterial** 既支持点光源又支持定向光源,而 **PhongPBMaterial** 仅支持点光源,这两者都不支持环境光源,而不像 **PhongBitmapMaterial**、**PhongColorMaterial** 和 **PhongMovieMaterial** 材质类支持环境光源。

在 **Away3D** 应用程序中选择使用何种光源类型,通常要由所选的材质确定,并不是由光源确定选用何种材质。

5.6 着色技术

Away3D 材质使用了很多着色技术,有时可以联合应用以完成最终的效果。可以使用这些技术把纹理用于 3D 对象表面上,用外部光源照亮 3D 对象,显示周边环境的反射光,或模拟凹凸不平的表面。

5.6.1 纹理映射

使用纹理映射,通常将 PNG、JPG 和 GIF 图像文件应用到 3D 对象的表面。可使用纹理映射显示简单的纹理,或与其他着色技术相结合。

图 5-2 显示了一个球体,对它使用了纹理映射,显示简单的代表地球的纹理。

5.6.2 法向贴图

法向贴图是一种把深度的表现添加到 3D 对象的表面技术,使用存储在图像里面的信

息调用法向贴图方法计算出如何着色材质的每个部分。着色能够表现一种崎岖不平表面的效果。

法向贴图有添加深部细节的优点而不必使用附加 polygons 模型。法向贴图一个低模 3D 对象通常比映射一个带标准材质的高模 3D 对象要快些,却可以维持高模 3D 对象的可视质量。

建立法向贴图的实用工具,在网站 <http://www.tartiflop.com/disp2norm/> 可以找到,这个工具用于建立能替换灰梯映射到平面 3D 对象或球面 3D 对象的法向贴图。

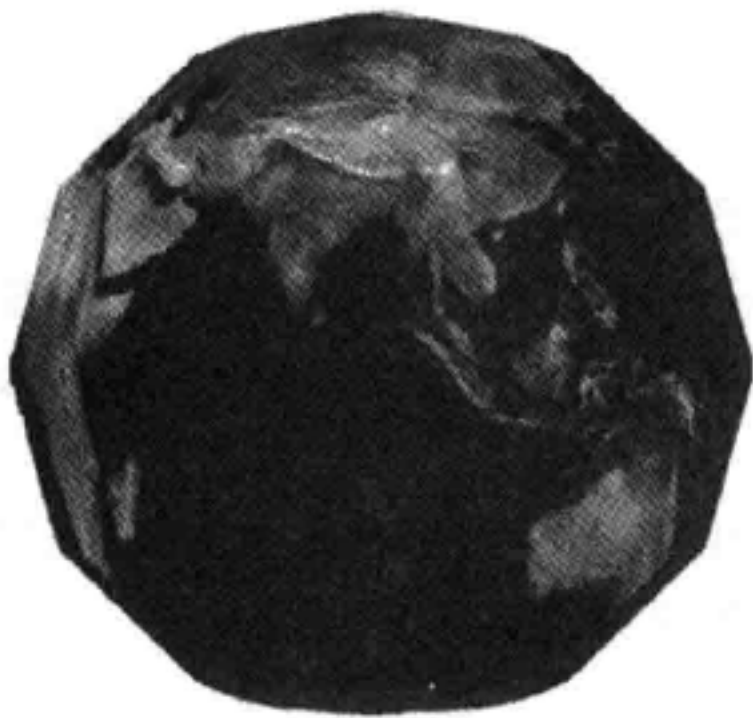


图 5-2 地球的纹理效果

图 5-3 是一个法向贴图应用到球面的例子。

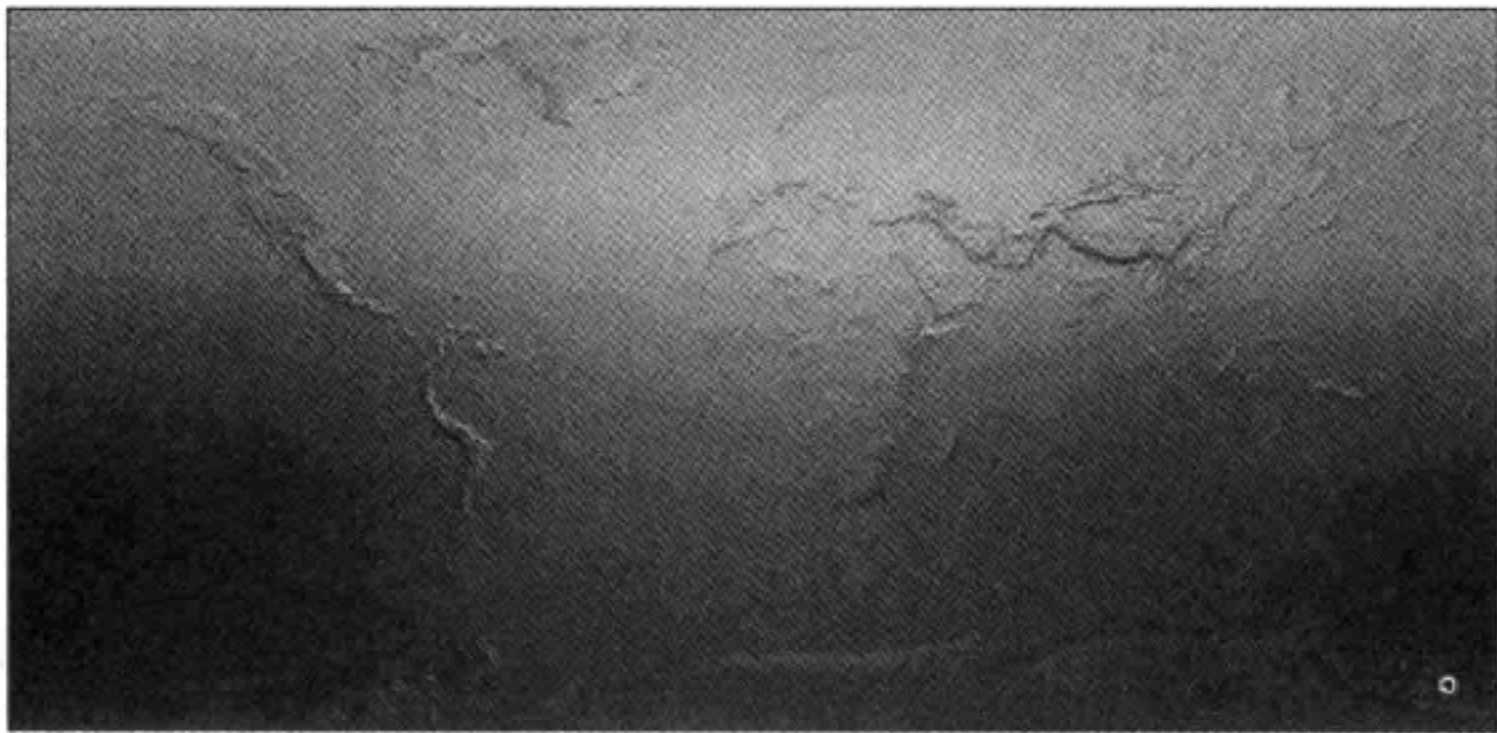


图 5-3 球面的法向贴图效果

图 5-4 展示了它的效果,可以看见球体显现出粗糙的表面,从屏幕截图的视角看,崎岖不平的效果在非洲大陆上特别突出。



图 5-4 法向贴图的地球纹理效果

5.6.3 环境映射

环境映射是一种用来画出 3D 对象表面对周围环境反射画面的技术,如果要画出在它的表面反射 3D 对象周边真实的环境的画面,要花费极高的计算代价。但是这种效果可用简单的纹理或显现 3D 对象周边的多个立体纹理来近似地显示。

环境映射对建立一个有光泽的 3D 对象外貌是非常有用的,如那些磨光的或是金属表面。在图 5-5 中,前两个 3D 对象应用了实现环境映射的材质(反射大理石纹理)。最左边的圆环体应用环境映射在底部纹理上面,而中间的圆环体应用环境映射在单色上面,作为对比,最右边的圆环体仅应用材质的纹理映射。

通过环境映射产生的效果用静态的屏幕截图显示出来是非常困难的,但是当 3D 对象相对于照相机运动时便立即表现出来了。



图 5-5 圆环体不同纹理映射的效果

5.6.4 平面着色

平面着色用于对着光源照明的 polygon 模型,它的计算速度非常快。由于作为一个整体,每个三角面都要着色到的原因,对于低模 3D 对象的三角面的边缘显得突出了一些。

如图 5-6 所示的球体,使用了平面着色,正如所看到的,很容易分辨出组成球体的三角面。

5.6.5 Phong 着色

Phong 着色用于计算对着光源的 3D 对象表面上每个像素点的照明度,它能消除平面着色产生的突然的边沿,但这是以运行成本为代价的。

如图 5-7 所示的球体使用了 Phong 着色,因为每个像素点不受三角面的约束而照明,最终导致比上述平面着色技术的表面要光滑很多。

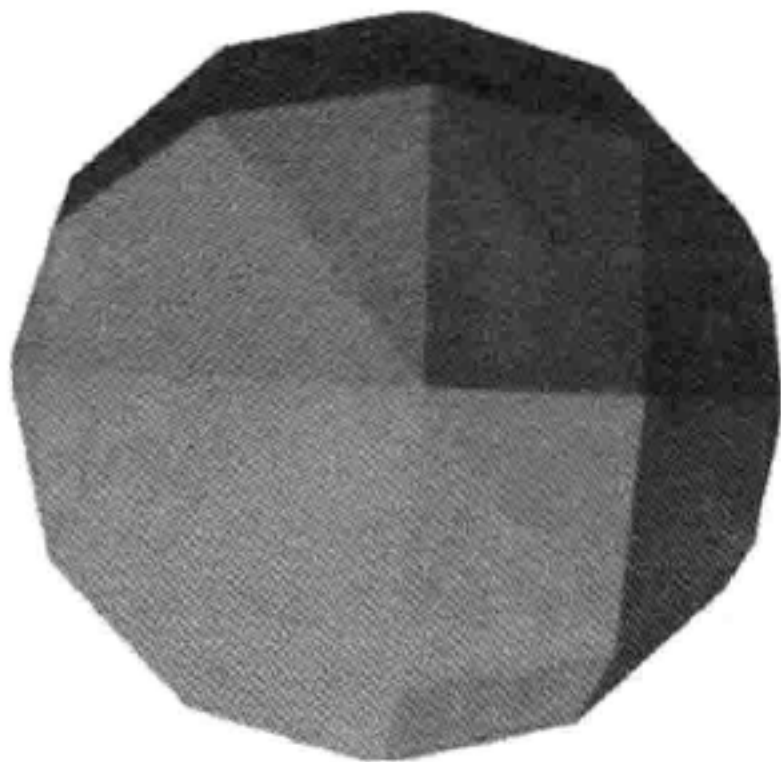


图 5-6 球体的平面着色效果



图 5-7 球体的 Phong 着色效果

5.7 应用材质

为了展示 Away3D 中这些基本材质的可用性,建立一个新的叫做 MaterialsDemo 的范例程序。

```
package  
{
```

某些原始模型显示材质比其他模型要醒目些,为此在这个例子里,把各种各样的材质应用于本程序里的球、圆环、立方体和平面的原始对象上。所有的原始对象都继承 Mesh 类,它能逻辑选择引用 4 个原始模型实例 instances 变量的类型。


```
import away3d.core.base.Mesh;
```

Cast 类提供了很多方便的函数,这些函数用于处理对象类型间的切换。

```
import away3d.core.utils.Cast;
```

前面已经讲过,有些被照明的材质支持点光源和定向光源(有的两者都支持),为突显照明的材质,点光源和定向光源都应添加到场景里来。

```
import away3d.lights.DirectionLight3D;  
import away3d.lights.PointLight3D;
```

为了能从外部装入纹理图像 images 文件,还要引入 TextureLoadQueue 和 TextureLoader 类。

```
import away3d.loaders.utils.TextureLoadQueue;  
import away3d.loaders.utils.TextureLoader;
```

还要引入本 MaterialsDemo 类展示的各种材质类,这些材质类来自 away3d.materials 包。

```
import away3d.materials.AnimatedBitmapMaterial;  
import away3d.materials.BitmapFileMaterial;  
import away3d.materials.BitmapMaterial;  
import away3d.materials.ColorMaterial;  
import away3d.materials.CubicEnvMapPBMaterial;  
import away3d.materials.DepthBitmapMaterial;  
import away3d.materials.Dot3BitmapMaterial;  
import away3d.materials.Dot3BitmapMaterialF10;  
import away3d.materials.EnviroBitmapMaterial;  
import away3d.materials.EnviroColorMaterial;  
import away3d.materials.FresnelPBMaterial;  
import away3d.materials.MovieMaterial;  
import away3d.materials.PhongBitmapMaterial;  
import away3d.materials.PhongColorMaterial;  
import away3d.materials.PhongMovieMaterial;  
import away3d.materials.PhongMultiPassMaterial;  
import away3d.materials.PhongPBMaterial;  
import away3d.materials.ShadingColormaterial;  
import away3d.materials.TransformBitmapMaterial;  
import away3d.materials.WhiteShadingBitmapMaterial;  
import away3d.materials.WireColorMaterial;  
import away3d.materials.WireframeMaterial;
```

这些材质将全部用于很多原始模型,这些原始模型要从 away3d.primitives 包引入。

```
import away3d.primitives.Cube;  
import away3d.primitives.Palme;  
import away3d.primitives.Sphere;  
import away3d.primitives.Torus;
```

CubeFaces 类定义了很多标识立方体 6 个面的常数。

```
import away3d.primitives.utils.CubeFaces;
```

当从外部图像 images 文件装入纹理时,要通过处理事件,在屏幕上显示有关 images 文件信息的文本域,为此要定义文本域的位置或在场景里的向量。还要用到下面的 Flash 类。

```
import flash.geom.Vector3D;
import flash.net.URLRequest;
import flash.display.BitmapData;
import flash.events.Event;
import flash.events.KeyboardEvent;
import flash.text.TextField;
```

MaterialsDemo 类继承了第 1 章中的 Away3DTemplate 类。

```
public class MaterialsDemo extends Away3DTemplate
{
```

在资源管理一节中,讨论管理资源的方法之一是嵌入方式。在这里我们看见,一个外部图像 JPG image 文件用 source 参数来引用,用 Embed 关键字嵌入进来。以这种方法嵌入 image 文件就表示建立一个实例化的 EarthDiffuse 类,它将使用 earth_diffuse.jpg 文件中包含的图像数据构成一个 Bitmap 对象。

```
[Embed(source="earth_diffuse.jpg")]
protected var EarthDiffuse:Class;
```

以同样的方法,嵌入更多的附加图像。

```
[Embed(source="earth_normal.jpg")]
protected var EarthNormal:Class;
[Embed(source="earth_specular.jpg")]
protected var EarthSpecular:Class;
[Embed(source="checkerboard.jpg")]
protected var Checkerboard:Class;
[Embed(source="bricks.jpg")]
protected var Bricks:Class;
[Embed(source="marble.jpg")]
protected var Marble:Class;
[Embed(source="water.jpg")]
protected var Water:Class;
[Embed(source="waternormal.jpg")]
protected var WaterNormal:Class;
[Embed(source="spheremap.jpg")]
protected var SphereMap:Class;
[Embed(source="skyleft.jpg")]
```

```
protected var Skyleft:Class;
[Embed(source="skyfront.jpg")]
protected var Skyfront:Class;
[Embed(source="skyright.jpg")]
protected var Skyright:Class;
[Embed(source="skyback.jpg")]
protected var Skyback:Class;
[Embed(source="skyup.jpg")]
protected var Skyup:Class;
[Embed(source="skydown.jpg")]
protected var Skydown:Class;
```

这里,还要嵌入三个 SWF 文件,这里的嵌入正如先前的 images 文件一样。

```
[Embed(source="Butterfly.swf")]
protected var Butterfly:Class;
[Embed(source="InteractiveTexture.swf")]
protected var InteractiveTexture:Class;
[Embed(source="Bear.swf")]
protected var Bear:Class;
```

TextField 对象用来在屏幕上显示当前材质的名字。

```
protected var materialText:TextField;
```

currentPrimitive 属性用于引用应用各种材质到它上面的原始模型。

```
protected var currentPrimitive:Mesh;
```

directionalLight 和 pointLight 属性,每个引用一个添加到场景里用来照明材质的光源。

```
protected var directionalLight:DirectionalLight3D;
protected var pointLight:PointLight3D;
```

当我们希望球沿 Z 轴弹跳时, bounce 属性设置为 true,用图像跳动时突出显示 DepthBitmapMaterial 材质类的效果。

```
protected var bounce:Boolean;
```

frameCount 属性保存当 bounce 属性设置为 true 时,帧已经画出的次数。

```
protected var frameCount:int;
```

构造函数 constructor 调用 Away3DTemplate 的构造函数,初始化 3D 引擎。

```
public function MaterialsDemo()
{
```

```
super();
}
```

removePrimitive()函数用于消除场景里当前的原始模型,为新建一个原始模型做准备。

```
protected function removePrimitive():void
{
    If (currentPrimitive != null)
    {
        scene.removeChild(currentPrimitive);
        currentPrimitive=null;
    }
}
```

initSphere()函数首先调用 removePrimitive()函数,消除场景里当前已有的原始组件,然后建立一个新的球体 sphere 原始组件,并添加到场景里。作为选项,可以把 bounce 属性设置为 true,这就使得球体可沿 Z 轴弹跳。

```
protected function initSphere(bounce:Boolean=false):void
{
    removePrimitive();
    currentPrimitive=new Sphere();
    scene.addChild(currentPrimitive);
    this.bounce=bounce;
}
```

initTorus()、initCube()和 initPlane()函数与 initSphere()函数具有一样的功能,把这里指定的原始模型添加到场景里,这些函数设置 bounce 属性为 false,使应用到这些原始模型的材质没有一个能使用场景里原来的 bounce 属性。

```
protected function initTorus():void
{
    removePrimitive();
    currentPrimitive=new Torus();
    scene.addChild(currentPrimitive);
    this.bounce=false;
}
protected function initCube():void
{
    removePrimitive();
    currentPrimitive=new Cube(
    {
        width:200,
        height:200,
        depth:200
    }
}
```



```
);  
scene.addChild(currentPrimitive);  
this.bounce=false;  
}  
  
protected function initPlane():void  
{  
    removePrimitive();  
    currentPrimitive=new Plane(  
        {  
            bothsides:true,  
            width:200,  
            height:200,  
            yUp:false  
        }  
    );  
    scene.addChild(currentPrimitive);  
    this.bounce=false;  
}
```

removeLights()函数用于消除场景里已有的光源,为新建一个新光源做准备。

```
protected function removeLights():void  
{  
    If (directionalLight != null)  
    {  
        scene.removeLight(directionalLight);  
        directionalLight = null;  
    }  
    If (pointLight != null)  
    {  
        scene.removeLight(pointLight);  
        pointLight = null;  
    }  
}
```

调用 initPointLight()函数和 initDirectonalLight()函数前首先调用 removeLights()函数,消除场景里已经存在的光源,然后建立新的代表各自的一个光源,并添加到场景里。

```
protected function initPointLight():void  
{  
    removeLights();  
    pointLight=new PointLight3D(  
        {  
            x: -300,  
            y: -300,
```

```

        radius: 1000
    }
);
scene.addLight(pointLight);
}
protected function initDirectionalLight():void
{
    removeLights();
    directionalLight=new DirectionalLight3D(
    {
        x: 300,
        y: 300,

```

默认的光源指向的方向设置是(0,0,0),这样表示光源没有指向任何方向。如果有一个不反射照明材质表面的定向光源,留下这个默认方向属性值。这里重载默认值,使光源指向原点。

```

        direction: new Vector3D(-1,-1,0)
    }
);
scene.addLight(directionalLight);
}

```

initScene()函数已经重载用于调用 applyWireColorMaterial()函数,这个函数将显示一个带有 WireColorMaterial 材质的球体。照相机的位置也设置回原点。

```

protected override function initScene():void
{
    super();
    this.camera.z=0;
    applyWireColorMaterial();
}

```

initUI()函数把文本域添加到舞台,文本域用于显示当前材质的名字。

把文本域 **TextField** 作为子对象添加到主类 **Sprite(Away3DTemplate 继承它)**里,还没有指定与 **Away3D** 引擎相关联,如此显示在屏幕上的 2D 的 **x、y** 坐标,还不是 **Away3D** 场景中的 3D 坐标。

```

protected override function initUI():void
{
    materialText=new TextField();
    materialText.x=10;

```

```

materialText.y=10;
materialText.width=300;
this.addChild(materialText);
}

```

initListeners()函数已经包含用于注册的 onKeyUp()函数,每当键盘上的一个键释放时,就要调用它。

```

protected override function initListeners():void
{
    super.initListeners();
    stage.addEventListener(KeyboardEvent.KEY_UP, omKeyUp);
}

```

onEnterFrame()函数已经包含驱动当前原始模型的功能。

```

protected override function onEnterFrame(event:Event):void
{
    super.onEnterFrame(event);
}

```

如果 bounce 属性设置为 true,则当前的原始模型将沿 Z 轴的 400~600 个单位之间跳动,用于突出显示 DepthBitmapMaterial 材质类产生的效果。

```

If (bounce)
{
    ++frameCount;
    currentPrimitive.z=500+Math.sin(frameCount/10)*100;
}

```

如果 bounce 属性设置为 false,当前的原始模型将固定在沿 Z 轴 500 个单位处。

```

else
{
    frameCount=0;
    currentPrimitive.z=500;
}

```

当前的原始模型也将缓慢地围绕 X 轴和 Y 轴转动。

```

currentPrimitive.rotationX += 1;
currentPrimitive.rotationY += 1;
}

```

onKeyUp()函数用 switch 语句调用相应的函数,响应键盘上释放某键产生的中断请求。紧挨着每条 case 语句的注释,写的是相对于键代码的关键字。

```

protected function onKeyUp(event:KeyboardEvent):void
{

```

```
switch(event.keyCode)
{
    case 49:      //1
        applyWireColorMaterial();
        break;
    case 50:      //2
        applyWireframeMaterial();
        break;
    case 51:      //3
        applyColorMaterial();
        break;
    case 52:      //4
        applyBitmapMaterial();
        break;
    case 53:      //5
        applyBitmapMaterial();
        break;
    case 54:      //6
        applyMovieMaterial();
        break;
    case 55:      //7
        applyInteractiveMovieMaterial();
        break;
    case 56:      //8
        applyAnimatedBitmapMaterial();
        break;
    case 57:      //9
        applyDot3BitmapMaterialF10();
        break;
    case 48:      //0
        applyDot3BitmapMaterial();
        break;
    case 81:      //Q
        applyEnviroBitmapMaterial();
        break;
    case 87:      //W
        applyEnviroColorMaterial();
        break;
    case 69:      //E
        applyFresnelPBMaterial();
        break;
    case 82:      //R
```



```

        applyPhongBitmapMaterial();
        break;
    case 84:    //T
        applyPhongColorMaterial();
        break;
    case 89:    //Y
        applyPhongMovieMaterial();
        break;
    case 85:    //U
        applyPhongPBMaterial();
        break;
    case 73:    //I
        applyPhongMultiPassMaterial();
        break;
    case 79:    //O
        applyShadingColorMaterial();
        break;
    case 80:    //P
        applyWhiteShadingBitmapMaterial();
        break;
    case 65:    //A
        applyTransformBitmapMaterial();
        break;
    case 83:    //S
        applyCubicEnvMapPBMaterial();
        break;
    case 68:    //D
        applyBitmapFileMaterial();
        break;
    case 70:    //F
        applyExternalDoc3BitmapMaterial();
        break;
    }
}

```

MaterialDemo 类余下的部分,就是要建立一些由 onKeyUp() 函数调用的功能函数,这些功能函数要建立原始 3D 模型对象,并把材质应用到它们上面,必要时还要创建光源。这些函数用的材质,以及功能函数本身,材质能接受的参数的相关介绍在随后的各节里。

不像第 2 章里建立原始 3D 对象,在那里,通常接受一个简单的初始化对象作为构造函数的参数。在 Away3D 材质类的构造函数中,有接受联合规范参数和初始对象这两种。为区分两者,在表 5-2 中,规范参数用黑体字表示。

5.8 基本材质

基本材质不需要纹理,也无须装入或嵌入外部资源,这使它们很容易被使用,因此是快速原型应用的主力。

5.8.1 线色彩材质

在前面演示的大多数例子中,没有指定特定的材质应用到 3D 对象上。当没有指定材质时,Away3D 将默认使用 WireColorMaterial 材质,它把 3D 对象着色成单色(这种颜色是程序运行时随机选择的,除非指定了颜色),并画出 3D 对象三角面的轮廓线,如图 5-8 所示。

这里,将特别地新建一个 WireColorMaterial 类的实例,并把它应用到 3D 对象上,材质的颜色用彩色名字的字符串指定。

```
protected function applyWireColorMaterial():void  
{
```



图 5-8 WireColorMaterial 材质着色效果

调用 initSphere() 函数,把球体原始组件 Sphere 添加到场景里。

```
initSphere();
```

材质的名字赋给文本域的 text 属性,它把材质的名字显示在屏幕上。

```
materialText.text = "WireColorMaterial";
```

然后创建材质自身并把它赋给本地变量 newMaterial。

这里,定义一个叫做 dodgerblue 的单色(使用串 string 定义颜色),而线框的颜色定义为白色(通过整数定义 0xFFFFFFFF),线宽为两个像素点。

```
var newMaterial:WireColorMaterial= new WireColorMaterial("dodgerblue",
{
    wirecolor: 0xFFFFFFFF,
    width:2
});
```

然后通过 material 属性,把新材质 newMaterial 赋给了原始组件 Sphere。

```
currentPrimitive.material= newMaterial;
```

表 5-2 列出了 WireColorMaterial 材质构造函数能接受的参数,那些黑体字参数直接传递到构造函数,而其他的参数则通过初始化对象传递到构造函数。

WireColorMaterial 类继承了 WireFrameMaterial 类,这就说明,WireFrameMaterial 类所列的初始化对象 ini object 的参数,也能用于这里的 WireColorMaterial 类。

表 5-2 WireColorMaterial 类构造函数的参数

参数	数据类型	默认值	说明
color	int / string	null / random	定义单色
alpha	Number	1	定义材质透明度
wirecolor	int / string	0x000000	定义线框颜色

5.8.2 线框材质

线框材质 WireframeMaterial 仅画出组成 3D 对象的三角面的轮廓线,如图 5-9 所示。在这个例子中,线框颜色用 int 指定,int 等价于应用在 applyWireColorMaterial 函数的 dodgerblue 的单色。

```
protected function applyWireframeMaterial():void
{
    initSphere();
    materialText.text="WireframeMaterial";
    var newMaterial:WireframeMaterial= new WireframeMaterial(0x1E90FF,
    {
        width:2
    });
    currentPrimitive.material= newMaterial;
}
```

表 5-3 列出了 WireframeMaterial 材质构造函数能接受的参数,那些黑体字参数直接传递到构造函数,而其他的参数则通过初始化对象传递到构造函数。

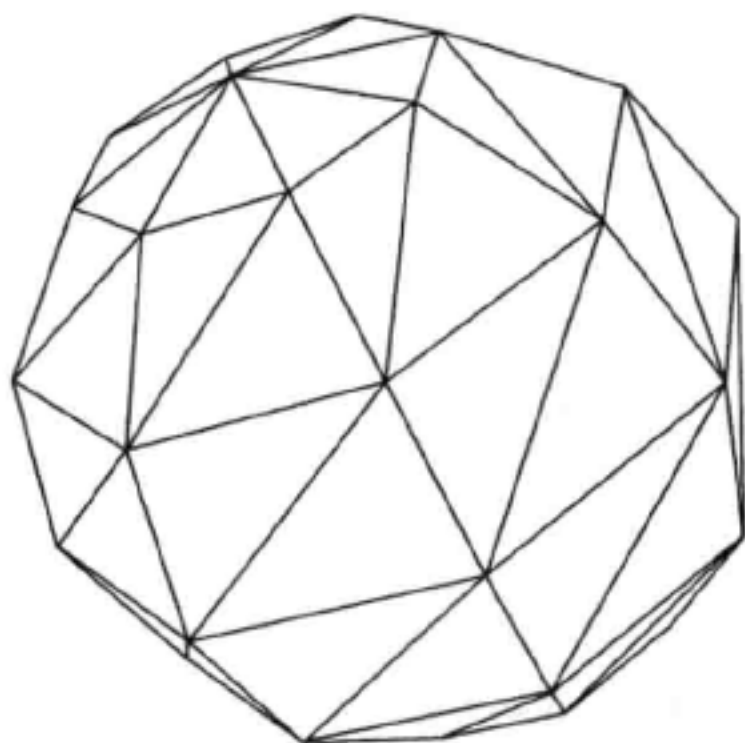


图 5-9 WireframeMaterial 材质着色效果

表 5-3 WireframeMaterial 类构造函数的参数

参数	数据类型	默认值	说明
wirecolor	int / string	null / random	定义线框颜色
wireAlpha	Number	1	定义材质透明度
thickness	Number	1	定义线框宽度

5.8.3 彩色材质

彩色材质 ColorMaterial 把一个单色应用到 3D 对象的表面,如图 5-10 所示。这个例子提供的颜色用与字符串 dodgerblue 版本等同的十六进制表示。

```
protected function applyColorMaterial():void
{
    initSphere();
    materialText.text="ColorMaterial";
    var newMaterial:ColorMaterial= new ColorMaterial("1E90FF");
    currentPrimitive.material= newMaterial;
}
```

ColorMaterial 类继承了 WireColorMaterial 类,这就说明凡是 WireColorMaterial 类所列出的初始化对象 ini object 的参数,也能用于 ColorMaterial 类。

如果初始化对象 init Object 的 debug 参数设置为 true, ColorMaterial 类的实例画的结果,就像 WireColorMaterial 类一样。表 5-4 列出了 ColorMaterial 材质构造函数能接受的参数,那些黑体字参数直接传递到构造函数,而其他的参数则通过初始化对象传递到构造函数。



图 5-10 ColorMaterial 材质着色效果

表 5-4 ColorMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
color	int / string	null / random	定义单颜色
debug	Boolean	false	当定义为 true 时,画的结果就像 WireColorMaterial 类一样

5.8.4 Bitmap 材质

Bitmap 材质在 3D 对象的表面上显示纹理,这些纹理通常用外部文件 PNG、JPG 和 GIF 定义,而这些文件是用图像编辑软件,如 Photoshop 建立的。

1. BitmapMaterial 类

BitmapMaterial 类把 Bitmap 纹理施加到 3D 对象的表面,如图 5-11 所示。在这个例子中,在 EarthDiffuse 类里包含 Bitmap 的数据,而 EarthDiffuse 类是从嵌入的 image 建立的。

```
protected function applyBitmapMaterial():void
{
    initSphere();
    materialText.text="BitmapMaterial";
}
```

这里,使用静态的 Cast 类里的 bitmap() 函数,当 BitmapMaterial 构造函数请求时,把 EarthDiffuse 类投放到 BitmapData 对象里。

```
var newMaterial:BitmapMaterial= new
    BitmapMaterial(Cast.bitmap(EarthDiffuse));
currentPrimitive.material= newMaterial;
}
```

Cast 类提供了很多方便的函数,在类型间进行投放,但是不使用 Cast 类的 bitmap()函数也是可以的。BitmapMaterial 类的新实例用下面的代码也能建立。

```
var newMaterial: BitmapMaterial = new BitmapMaterial ( new EarthDiffuse ( ).
    bitmapData);
```

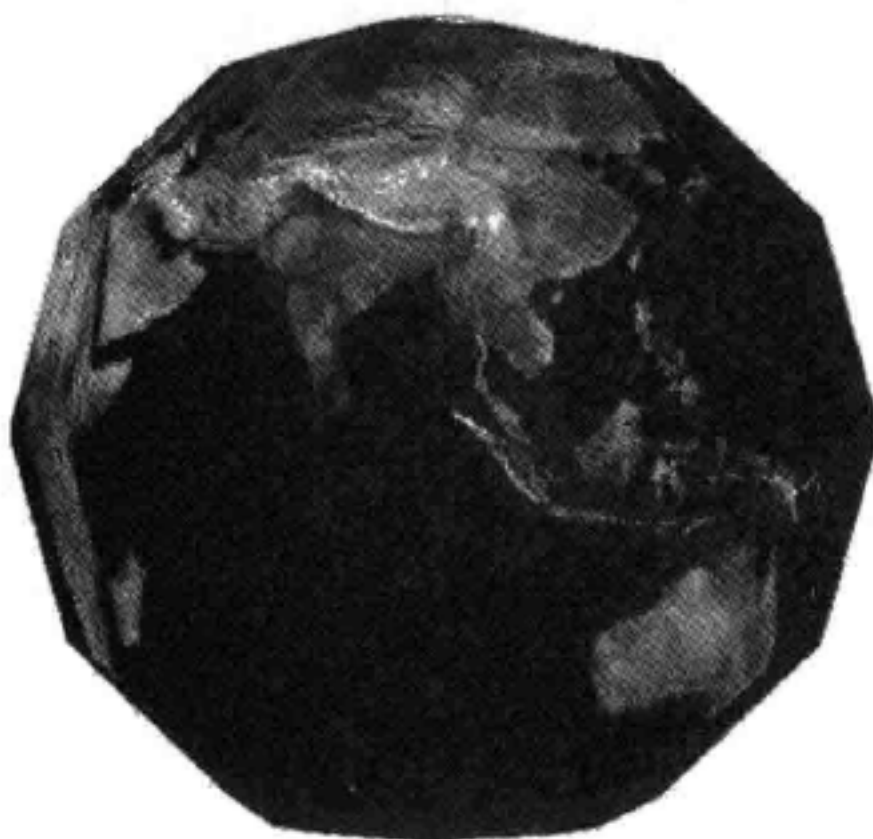


图 5-11 BitmapMaterial 材质效果

表 5-5 列出了 BitmapMaterial 材质构造函数能接受的参数,那些黑体字参数直接传递到构造函数,而其他的参数则通过初始化对象传递到构造函数。

表 5-5 BitmapMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		定义 bitmapData 对象,用作材质的纹理
. wireColor	int / string	0x000000	当 debug 设置为 true 时,定义三角形轮廓颜色
smooth	Boolean	false	画出时,确定纹理的 Bitmap 是否光滑(双线性过滤)
debug	Boolean	false	设为 true,画出带白色轮廓线的三角形,否则,精确画出带蓝色轮廓线的三角形
repeat	Boolean	false	确定纹理 Bitmap 是否用 uv-space 平铺
blendMode	String	BlendMode, NORMAL	为纹理 Bitmap 定义 blendMode 的值,BlendMode 类是 Flash 的显示包,定义是否能用混合模式
colorTransform	ColorTransform	null	定义纹理的 Bitmap 色彩转换
showNormals	Boolean	false	显示正常的每个面
color	int / string	0xFFFFFFFF	定义覆盖 Bitmap 纹理的颜色

2. TransformBitmapMaterial 类

一个 Bitmap 材质要依据材质的 UV 坐标系统,应用到 3D 对象的表面。TransformBitmapMaterial 材质有很多属性,它能在 UV 坐标系统空间里使材质显现成缩放、偏置和旋转。

这里,使用 rotation 初始化对象参数,使材质旋转 45° ,旋转的效果将相当清晰地显示在立方体上。

当转换材质时,最可能设置 repeat 初始化对象参数为 true,这就确保了在横跨 3D 对象表面时,纹理能重复使用。如果 repeat 初始化对象参数设置为 false(这是默认值),纹理的转换和使用只能一次,不能重复使用,以致纹理的边缘拉伸到横跨其余 3D 对象的表面。图 5-12 左边的图形显示了 repeat 设置为 true 的结果,右边的图形显示了 repeat 设置为 false 的结果。

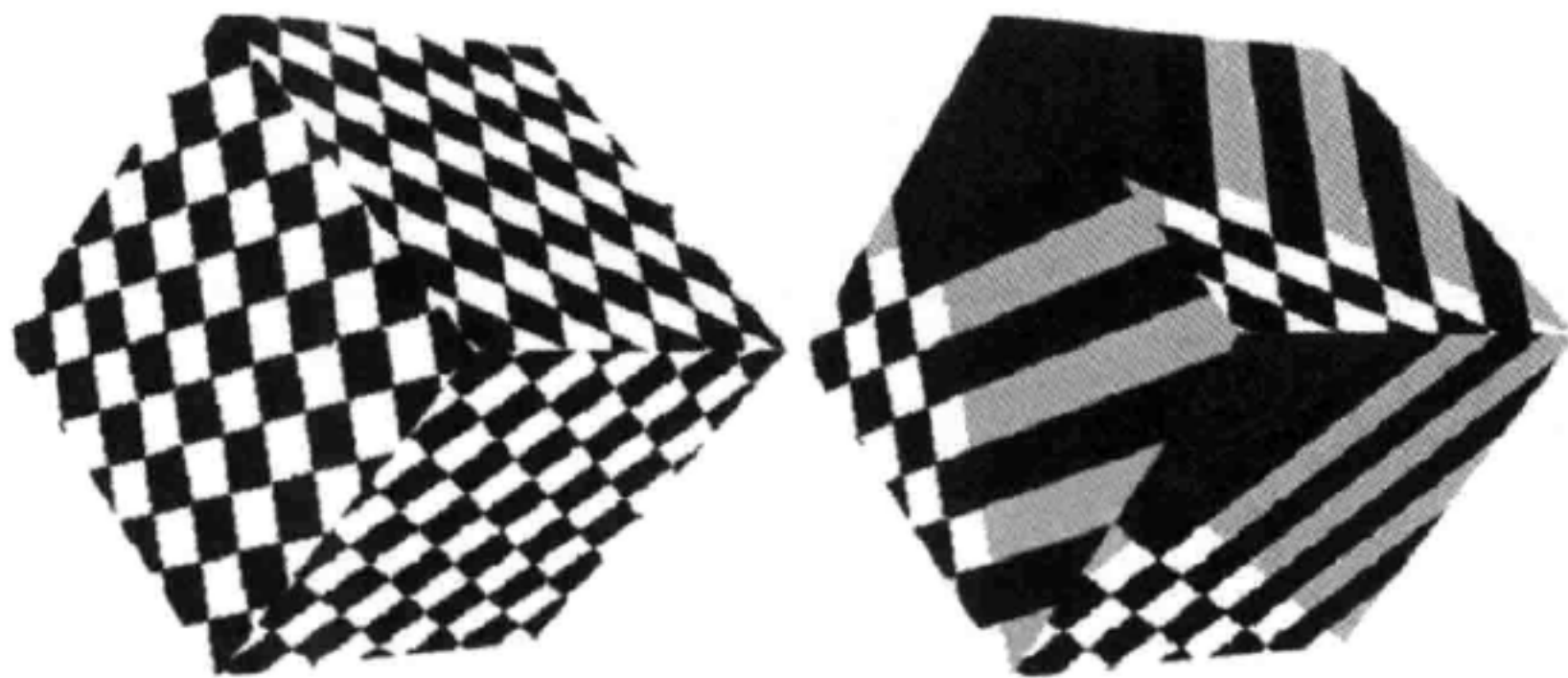


图 5-12 TransformBitmapMaterial 材质效果

```
protected function applyTransformBitmapMaterial():void
{
    initCube();
    materialText.text="TransformBitmapMaterial";
    var newMaterial: TransformBitmapMaterial = new TransformBitmapMaterial (Cast. bitmap
(Checkerboard),
    {
        repeat: true,
        rotation: 45
    }
    );
    urrentPrimitive.material= newMaterial;
}
```

TransformBitmapMaterial 类继承了 BitmapMaterial 类,这就说明,除表 5-6 所列出的

这些参数外,列在 BitmapMaterial 类的初始化对象参数,对 TransformBitmapMaterial 类也是有效的。

表 5-6 TransformBitmapMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		定义 bitmapData 对象,用作材质的纹理
transform	Matrix	null	用 UV 空间转换纹理
scaleX	Number	1	用 UV 空间缩放纹理的 x 坐标
scaleY	Number	1	用 UV 空间缩放纹理的 y 坐标
repeat	Boolean	false	确定纹理 Bitmap 是否用 uv-space 平铺
offsetX	Number	0	用 UV 空间偏置纹理的 x 坐标
offsetY	Number	0	用 UV 空间偏置纹理的 y 坐标
rotation	Number	0	用 UV 空间旋转纹理
projectionVector	Number3D	null	在对象空间投射纹理,忽略 UV 坐标系统的点对象。当设置为 null 时,纹理正常画出
throughProjection	Boolean	true	确定投射的纹理是否在指向远离投射点的面上可见
globalProjection	Boolean	false	确定投射的纹理是否使用 offsetX 和 offsetY 坐标,以及投射的向量相对场景坐标的值

5.8.5 动画材质

有很多材质能用于在 3D 对象的表面上显示动画,这些动画通常是被编码到 SWF 文件里的影视。也可将一个交互式的 SWF 文件显示在 3D 对象表面上,例如表单。

1. 影视材质

影视材质 MovieMaterial 显示 Sprite 对象的输出,Sprite 对象能够是动画,Sprite 对象源自另一个 SWF 文件来。在这种情况下,可嵌入并通过 Bear 类引用这个文件,然后一个新 Bear 类的实例被传递给 MovieMaterial 材质的构造函数。

```
protected function applyMovieMaterial():void
{
    initCube();
    materialText.text = " MovieMaterial";
    var newMaterial: MovieMaterial = new MovieMaterial (new Bear());
    currentPrimitive.material = newMaterial;
}
```

如图 5-13 所示为 MovieMaterial 材质效果。

MovieMaterial 类继承了 TransformBitmapMaterial 类,这就说明,除表 5-7 所列出的这

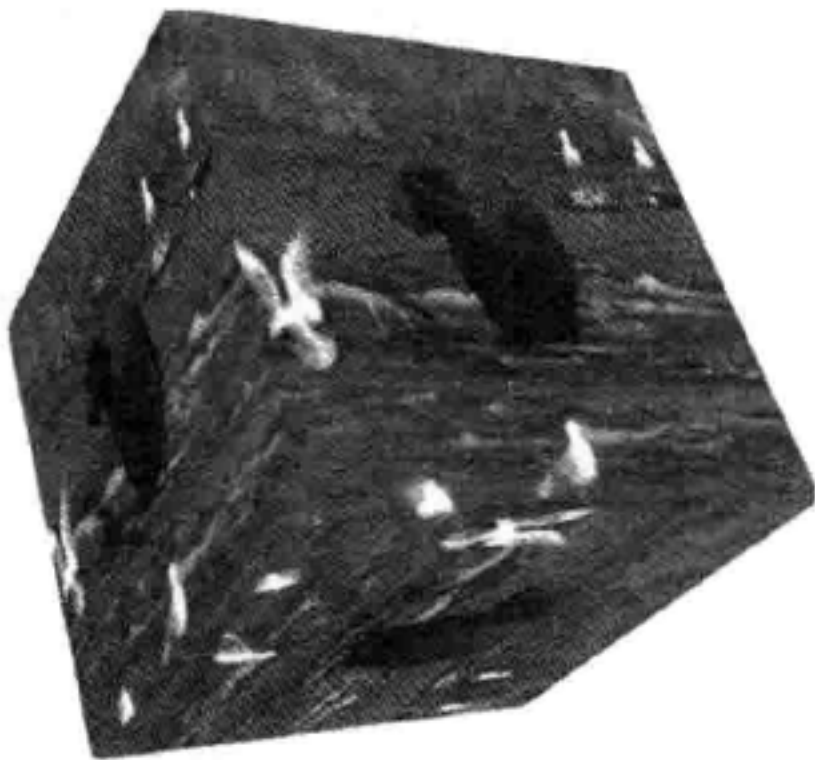


图 5-13 MovieMaterial 材质效果

些参数外,列在 TransformBitmapMaterial 类的初始化对象参数,用于 MovieMaterial 类也是有效的。

表 5-7 MovieMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
movie	Sprite		定义 Sprite 对象,用作材质的纹理
transparent	Boolean	true	建立动画时,定义纹理 Bitmap 的透明属性
autoUpdate	Boolean	true	指定每帧上纹理 Bitmap 是否更新
interactive	Boolean	false	指定材质是否把鼠标交互传到动画
lockW	int	Movie. width	锁定画出范围的宽度,而不是动画剪辑源
lockH	int	Movie. height	锁定画出范围的高度,而不是动画剪辑源

2. 动画 Bitmap 材质

AnimatedBitmapMaterial 类显示来自 MovieClip 对象的很多帧,为了提升运行性能。它首先把由 MovieClip 对象提供的每个帧画到 Bitmap 中,这些 Bitmaps 保存在缓冲区里,这样就提升了回放的运行性能,但是这是以增加内存成本为代价的。

因为缓冲区需要内存开销,AnimatedBitmapMaterial 不能用于长度超过 2s 的动画剪辑放映。如果传递超过 2s 的动画剪辑,将会抛出一个异常。

MovieClip 对象被传递到 AnimatedBitmapMaterial 的构造函数,它通常引自另一个 SWF 文件来,这个 SWF 源文件必须用 ActionScript 虚拟机 2(AVM2)格式来实现,这种格式用于 Flash Player 9 及以上版本,这点是很重要的,因为大量的视频版本工具输出的是 AVM1 SWF 文件。

如果必须用 AVM1 格式显示 SWF 视频,就要用 MovieMaterial 类来取代。

如果试图在 AnimatedBitmapMaterial 类里,使用 AVM1 格式的 SWF 文件,将会抛出一个类似下面的异常信息:

```
TypeError:Error # 1034: Type Coercion failed: cannot convert flash.display:AVM1Movie@51e8e51 to flash.display.MovieClip.
```

FFmpeg 是一个免费的跨平台的工具,它能够把视频文件转换成 AVM2 格式的 SWF 文件。可以从 <http://ffmpeg.org/> 下载并解压该软件。Windows 版的二进制文件,可从 <http://sourceforge.net/projects/mplayer-win32/files/FFmpeg/> 下载。以下的命令将把 WNV 视频文件变换成 2s 长的 SWF2 格式的文件,分辨率为 320×240,没有音频。

```
Ffmpeg -i Butterfly.wmv -t 2 -s 320x240 -an -f avm2 Butterfly.swf
```

```
protected function applyAnimatedBitmapMaterial():void
{
    initCube();
    materialText.text=" AnimatedBitmapMaterial";
    var newMaterial: AnimatedBitmapMaterial= new AnimatedBitmapMaterial (new Butterfly());
    currentPrimitive.material= newMaterial;
}
```

AnimatedBitmapMaterial 类继承了 TransformBitmapMaterial 类,这就说明,除表 5-8 所列出的这些参数外,列在 TransformBitmapMaterial 类的初始化对象参数,用于 AnimatedBitmapMaterial 类也是有效的。

表 5-8 AnimatedBitmapMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
movie	MovieClip		为 MovieClip 建立 Bitmap 缓冲,用于材质
loop	Boolean	true	指定动画是否循环
autoPlay	Boolean	true	指定动画是否在初始时启动放映
_index	int	0	设置帧索引

3. 交互式影视材质

通过把 interactive 参数设置为 true,MovieMaterial 对象能够将鼠标的事件传递到显示

的 Sprite,这允许用户与材质交互,犹如把交互面直接加到了 Flash 舞台一样。而交互材质是围绕包裹着 3D 对象的。

```
protected function applyInteractiveMovieMaterial():void
{
    initCube();
    materialText.text=" MovieMaterial--Interactive";
    var newMaterial: MovieMaterial = new MovieMaterial (new InteractiveTexture(),
    {
        interactive: true,
        smooth: true
    }
    );
    currentPrimitive.material= newMaterial;
}
```

如图 5-14 所示为交互式 MovieMaterial 材质效果。



图 5-14 交互式 MovieMaterial 材质效果

交互式动画材质 MovieMaterial 类的构造函数参数,参考表 5-7。

5.8.6 复合材质

复合材质是将两种或多种基类材质联合在一起,完成它们最终表现的样子。复合材质用于显示很多种不同的效果,如底纹、凹凸映射、环境映射、照明。

1. 深度 Bitmap 材质

DepthBitmapMaterial 材质与 BitmapMaterial 材质相似,它把 Bitmap 纹理用于 3D 对象的表面。除此之外,DepthBitmapMaterial 材质还根据 3D 对象到照相机的距离,对 3D 对象的表面着色。它能实现像烟雾状的效果。

```
protected function applyDepthBitmapMaterial():void
{
```

展示 DepthBitmapMaterial 类产生效果的最好例子,是相对照相机运动的 3D 对象。设置 initSphere() 的参数 bounce 为 true,这将引发 onEnterFrame() 函数使球体沿 Z 轴跳动。

```
initSphere(true);
materialText.text=" DepthBitmapMaterial";
var newMaterial: DepthBitmapMaterial= new DepthBitmapMaterial (Cast.bitmap(EarthDiffuse)
{
```

这里,定义画有基类 Bitmap 的纹理离照相机的距离,当材质靠近照相机近些,比 minZ 还小时,就要用 minColor 参数定义的颜色;当材质离照相机远些,比 maxZ 还大时,就要用 maxColor 参数定义的颜色。

```
minZ:400,
minColor:0xFFFFFFFF,
maxZ:500,
maxColor:0xFF000000
}
);
currentPrimitive.material= newMaterial;
}
```

如图 5-15 所示为 DepthBitmapMaterial 材质效果。



图 5-15 DepthBitmapMaterial 材质效果

DepthBitmapMaterial 是个复合材质,这就意味着它是用两个或多个基类材质完成最后的表现的,BitmapMaterial 是基类材质之一,为 DepthBitmapMaterial 提供初始化的对象,也被传送到 BitmapMaterial 的构造函数,这就说明,除表 5-9 所列出的这些参数外,列在 BitmapMaterial 的初始化对象参数,用于 DepthBitmapMaterial 类也是有效的。

表 5-9 DepthBitmapMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		bitmapData 对象用于材质的纹理
minZ	Number	500	最小 Z 向深度映射系数
maxZ	Number	2000	最大 Z 向深度映射系数
minColor	int	0xFFFFFFFF	最小 Z 向着色系数
maxColor	int	0x000000	最大 Z 向着色系数

2. 环境 Bitmap 材质

EnviroBitmapMaterial 类完成表面反射出周围图像的样子,其方法是用第二个 Bitmap 图像作为环境,映射到基类的 BitmapMaterial 材质上。

```
protected function applyEnviroBitmapMaterial():void
{
    initTorus();
    materialText.text="EnviroBitmapMaterial";
    var newMaterial: EnviroBitmapMaterial = new
        EnviroBitmapMaterial (
            Cast.bitmap(Bricks),
            Cast.bitmap(Marble)
        );
    currentPrimitive.material= newMaterial;
}
```

如图 5-16 所示为 EnviroBitmapMaterial 材质效果。



图 5-16 EnviroBitmapMaterial 材质效果

与 DepthBitmapMaterial 材质一样,EnviroBitmapMaterial 材质也是个复合材质,它把初始对象传送到包含 BitmapMaterial 类的实例,这就说明,除表 5-10 所列出的这些参数外,列在 BitmapMaterial 的初始化对象参数,用于 EnviroBitmapMaterial 类也是有效的。

表 5-10 EnviroBitmapMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		bitmapData 对象用于材质的纹理
enviroMap	BitmapData		bitmapData 对象用于材质的正常映射
mode	String	linear	设置映射方法的可能性,不影响环境材质类
reflectiveness	Number	0.5	材质反射系数

3. 环境彩色材质

EnviroColorMaterial 除了用单色材质而不是用 Bitmap 作为纹理之外,其他与 EnviroBitmapMaterial 类相同。

```
protected function applyEnviroColorMaterial():void
{
    initTorus();
    materialText.text="EnviroColorMaterial";
    var newMaterial: EnviroColorMaterial = new EnviroColorMaterial (
        "sandybrown",
        Cast.bitmap(Marble)
    );
    currentPrimitive.material= newMaterial;
}
```

如图 5-17 所示为 EnviroColorMaterial 材质效果。



图 5-17 EnviroColorMaterial 材质效果

EnviroColorMaterial 类直接继承了 ColorMaterial 类,这就说明,除表 5-11 所列出的这些参数外,列在 ColorMaterial 的初始化对象参数,也可用于 EnviroColorMaterial 类。

表 5-11 EnviroColorMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
color	int / string	random	字符串,十六进制值或彩色名。代表材质彩色
enviroMap	BitmapData		用 Bitmap 对象作为材质 map
mode	String	linear	设置是否映射方法的可能性,这个值不影响 EnviroColorMaterial
reflectiveness	Number	0.5	环境映射的反射系数
smooth	Boolean	false	当画到屏幕时,Bitmap 是否光滑(线性过滤)
color	int / string	0xFFFFFFFF	定义用到纹理上的色彩
blendMode	string	BlendMode.NORMAL	定义着色 Bitmap 的 BlendMode 值

5.8.7 光源材质

光源材质能用外部光源照明,正如之前讲过的,有三种光源能用于材质上,这些光源是:环境、点和定向光源。在光源与材质一节里,列出了哪些材质能用于哪些光源。

1. 白色底纹 Bitmap 材质

白色底纹 Bitmap 材质 WhiteShadingBitmapMaterial 类,用单调的底纹覆盖在照亮的 Bitmap 纹理上。正如这个类的名字一样,它的照明总是白色的而不管光源是什么颜色。

```
protected function applyWhileShadingBitmapMaterial():void
{
    initSphere();
    initPointLight();
    materialText.text = "WhileShadingBitmapMaterial";
    var newMaterial: WhileShadingBitmapMaterial = new WhileShadingBitmapMaterial (
        Cast.bitmap(EarthDiffuse)
    );
    currentPrimitive.material = newMaterial;
}
```

如图 5-18 所示为 WhiteShadingBitmapMaterial 材质效果。

WhiteShadingBitmapMaterial 类继承了 BitmapMaterial 类,这就说明,除表 5-12 所列出的这些参数外,列在 BitmapMaterial 类的初始化对象参数,用于 WhiteShadingBitmapMaterial 类也是有效的。

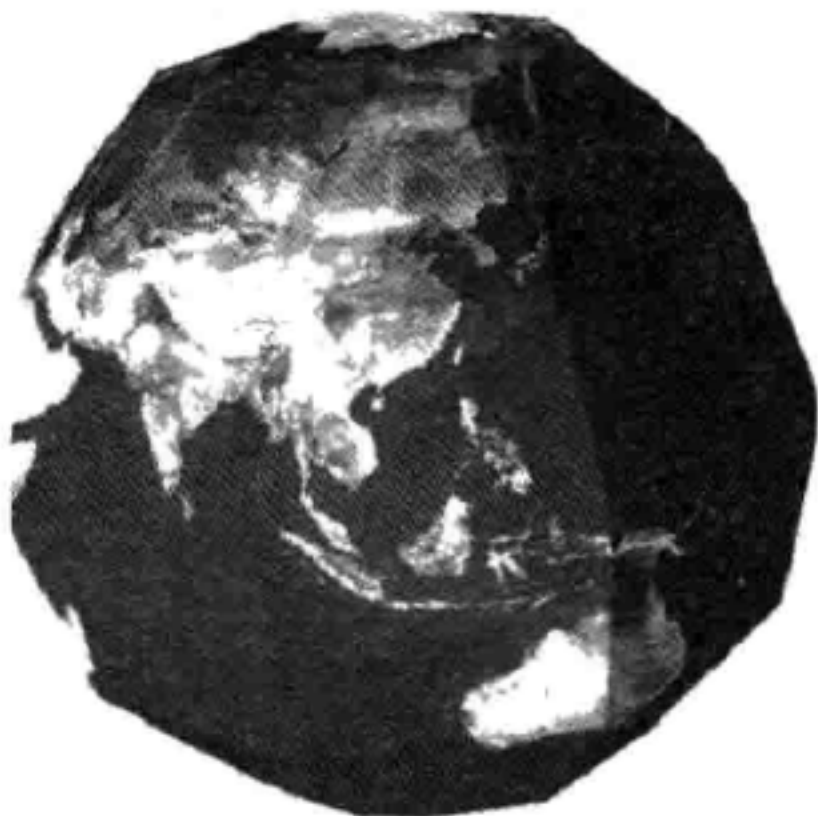


图 5-18 WhiteShadingBitmapMaterial 材质效果

表 5-12 WhiteShadingBitmapMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		bitmapData 对象用于材质的纹理
shininess	Number	20	发亮系数

2. 彩色底纹材质

彩色底纹材质 ShadingColorMaterial 用单调的底纹覆盖在单色的基类彩色上。

```
protected function applyShadingColorMaterial():void
{
    initSphere();
    initPointLight();
    materialText.text="ShadingColorMaterial";
    var newMaterial: ShadingColorMaterial= new
        ShadingColorMaterial (
            Cast.trycolor("deepkyblue")
        );
    currentPrimitive.material= newMaterial;
}
```

如图 5-19 所示为 ShadingColorMaterial 材质效果。

ShadingColorMaterial 类直接继承了 ColorMaterial 类,这就说明,除表 5-13 所列出的这些参数外,列在 ColorMaterial 的初始化对象参数,也可用于 ShadingColorMaterial 类。

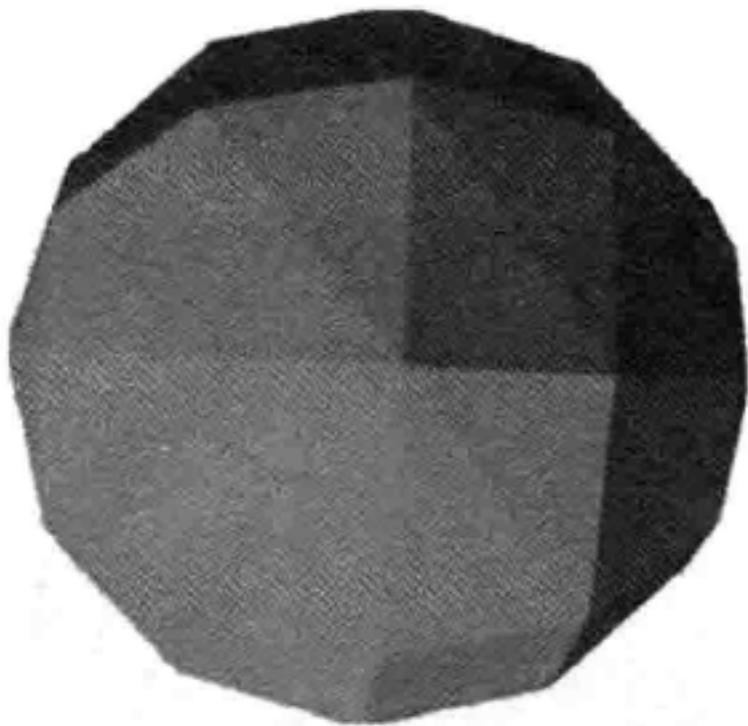


图 5-19 ShadingColorMaterial 材质效果

color 的参数可以是 int 型和 string 型,但由于 ColorMaterial 类有 bug,它仅能用 int 值才能正常工作。在上面的例子中,使用了 Cast 类的 trycolor() 函数,把代表彩色的串 deepkyblue 变换成 int 类型。

表 5-13 ShadingColorMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
color	int / string	random	字符串,十六进制值或彩色名。代表材质彩色
ambient	int / string	colorvalue	定义外界光颜色
diffuse	int / string	colorvalue	定义漫射光颜色
specular	int / string	colorvalue	定义反射光颜色
alpha	Number	1	定义材质的 Alpha 值
cache	Boolean	false	定义是否使色彩缓冲

3. PhongBitmap 材质

PhongBitmapMaterial 使用 Phong 底纹应用到 TransformBitmapMaterial 基类的材质上。

```
protected function applyPhongBitmapMaterial():void
{
    initSphere();
    initDirectionalLight();
    materialText.text="PhongBitmapMaterial";
    var newMaterial: PhongBitmapMaterial = new PhongBitmapMaterial ( Cast. bitmap
(EarthDiffuse)
```

```
    );  
    currentPrimitive.material= newMaterial;  
}
```

如图 5-20 所示为 PhongBitmapMaterial 材质效果。



图 5-20 PhongBitmapMaterial 材质效果

PhongBitmapMaterial 是一个复合材质,把初始化对象传送到包含 TransformBitmapMaterial 的实例,这就说明,除表 5-14 所列出的这些参数外,列在 TransformBitmapMaterial 的初始化对象参数,用于 PhongBitmapMaterial 类也是有效的。

表 5-14 PhongBitmapMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		bitmapData 对象用于材质的纹理
shininess	Number	20	用于突亮反射光的指数下降值
specular	Number	0.7	反射光级别系数

4. Phong 彩色材质

PhongColorMaterial 材质使用 Phong 底纹照明单色的基类材质。

```
protected function applyPhongColorMaterial():void  
{  
    initSphere();  
    initDirectionalLight();  
    materialText.text="PhongColorMaterial";  
    var newMaterial: PhongColorMaterial= new PhongColorMaterial ("deepskyblue");  
    currentPrimitive.material= newMaterial;  
}
```

图 5-21 为 PhongColorMaterial 材质效果。

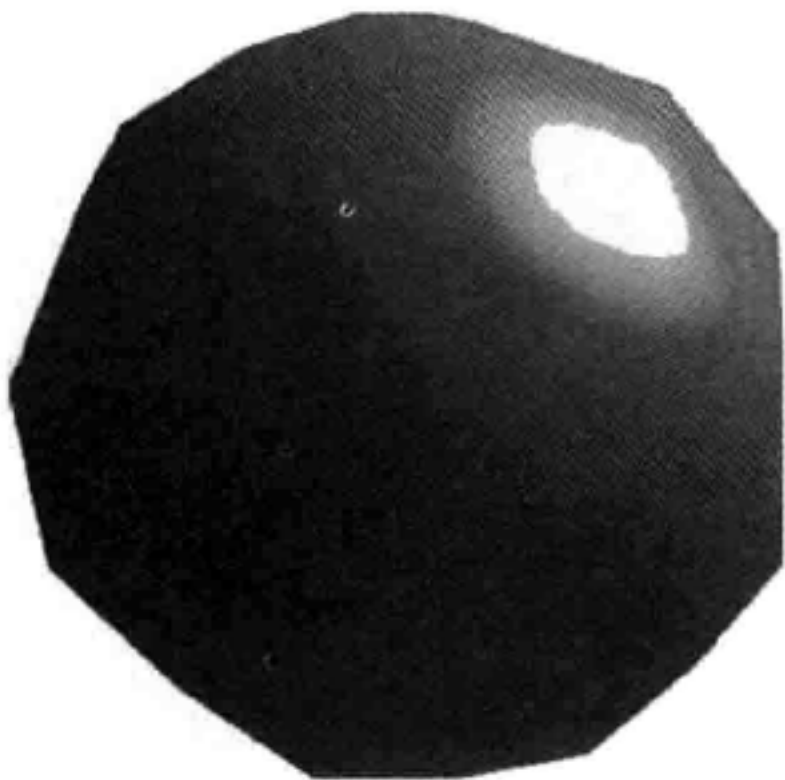


图 5-21 PhongColorMaterial 材质效果

表 5-15 列出了 PhongColorMaterial 材质构造函数能接受的参数,那些黑体字参数直接传递到构造函数,而其他的参数则通过初始化对象传递到构造函数。

表 5-15 PhongColorMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
color	int / string	0xFFFFFFFF	字符串,十六进制值或彩色名。代表材质颜色
shininess	Number	20	用于突亮反射光的指数下降值
specular	Number	0.7	反射光级别系数

5. PhongMovie 材质

PhongMovieMaterial 材质使用 Phong 着色照明动画的 MovieMaterial 基类的材质上。

```
protected function applyPhongMovieMaterial():void
{
    initSphere();
    initDirectionalLight();
    materialText.text="PhongMovieMaterial";
    var newMaterial: PhongMovieMaterial= new PhongMovieMaterial (new Bear());
    currentPrimitive.material= newMaterial;
}
```

如图 5-22 所示为 PhongMovieMaterial 材质效果。

PhongMovieMaterial 是一个复合材质,把初始化对象传送到包含 MovieMaterial 类的实例,这就说明,除表 5-16 所列出的这些参数外,列在 MovieMaterial 的初始化对象参数,用于 PhongMovieMaterial 类也是有效的。



图 5-22 PhongMovieMaterial 材质效果

表 5-16 PhongMovieMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
Movie	Sprite		Movie Clip 用于材质的纹理
shininess	Number	20	用于突亮反射光的指数下降值
specular	Number	0.7	反射光级别系数

6. Dot3Bitmap 材质

Dot3BitmapMaterial 材质使用法向贴图加深 3D 对象。

```
protected function applyDot3BitmapMaterial():void
{
    initSphere();
    initDirectionalLight();
    materialText.text = "Dot3BitmapMaterial";
    var newMaterial: Dot3BitmapMaterial = new Dot3BitmapMaterial (
        Cast.bitmap(EarthDiffuse),
        Cast.bitmap(EarthNormal)
    );
    currentPrimitive.material = newMaterial;
}
```

如图 5-23 所示为 Dot3BitmapMaterial 材质效果。

Dot3BitmapMaterial 是一个复合材质,把初始化对象传送到包含 BitmapMaterial 类的实例,这就说明,除表 5-17 所列出的这些参数外,列在 BitmapMaterial 的初始化对象参数,用于 Dot3BitmapMaterial 类也是有效的。



图 5-23 Dot3BitmapMaterial 材质效果

表 5-17 Dot3BitmapMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		bitmapData 对象,用于材质的纹理
normalMap	BitmapData		bitmapData 对象,用于材质的 DOT3 映射
shininess	Number	20	用于突亮反射光的指数下降值
specular	Number	0.7	反射光级别系数

5.8.8 Pixel Bender 材质

Away3D 包含很多用 Pixel Bender 制作的材质,这些材质时常产生比迄今为止所有材质更高的细致度的效果,使用 Pixel Bender 是 Away3D 第 3 版优于 Away3D 第 2 版的一大优点,这是因为 Away3D 第 3 版是针对 Flash Player 10 的,而 Away3D 第 2 版是针对 Flash Player 9 的。

1. Dot3BitmapMaterialF10 材质

Dot3BitmapMaterialF10 类是材质 Dot3BitmapMaterial 类的 Pixel Bender 版。

```
protected function applyDot3BitmapMaterial F10():void
{
    initSphere();
    initDirectionalLight();
    materialText.text="Dot3BitmapMaterialF10";
    var newMaterial: Dot3BitmapMaterialF10= new Dot3BitmapMaterialF10 (
        Cast.bitmap(EarthDiffuse),
        Cast.bitmap(EarthNormal)
    );
}
```

```

        currentPrimitive.material = newMaterial;
    }

```

Dot3BitmapMaterialF10 类继承了 BitmapMaterial 材质类,这就说明,列在 BitmapMaterial 的初始化对象的参数,用于 Dot3BitmapMaterialF10 类也是有效的。

如图 5-24 所示为 Dot3BitmapMaterialF10 材质效果。

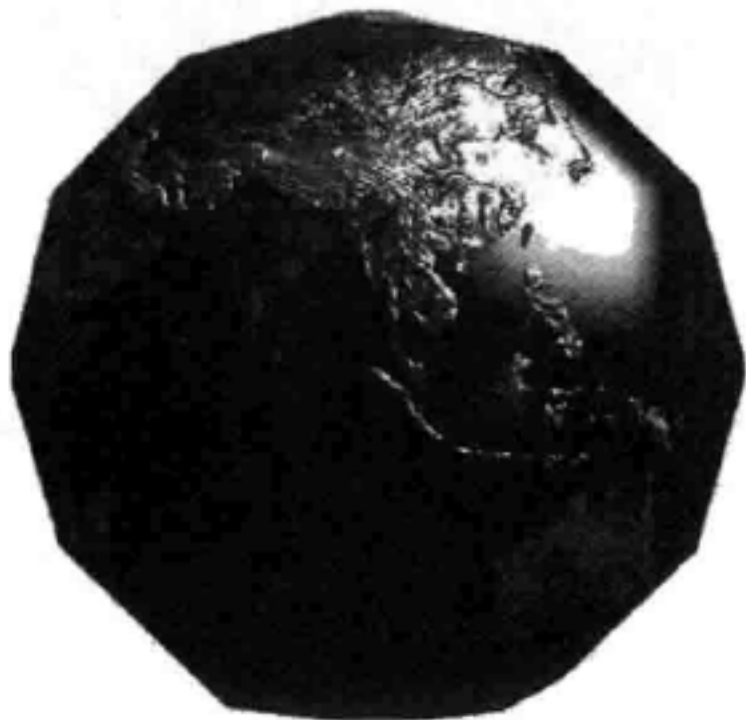


图 5-24 Dot3BitmapMaterialF10 材质效果

用于 Bitmap 的纹理和 normalMap 的参数必须有相同的维数。表 5-18 列出了 Dot3BitmapMaterialF10 类构造函数所需的参数。

表 5-18 Dot3BitmapMaterialF10 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		bitmapData 对象,用于材质的纹理
normalMap	BitmapData		bitmapData 对象,用于材质的 DOT3 映射
shininess	Number	20	用于突亮反射光的指数下降值
specular	Number	0.7	反射光级别系数

2. PhongPBMaterial 材质

PhongPBMaterial 类使用 Phong 底纹和法向贴图,给 3D 对象添加深度和照明,此外,它用镜面映射定义了反射光的强度,使亮的区域比暗区反射更多的光线。

图 5-25 嵌入在叫做 EarthSpecular 类的应用程序中,海洋显示得亮一些,这表示突显这一区域的反射光,大陆板块显示得暗一些,它挡住了来自亮区的反射光线。



图 5-25 应用于 PhongPBMaterial 法向贴图的图像

```
protected function applyPhongPBMaterial():void
{
    initSphere();
    initPointLight();
    materialText.text=" PhongPBMaterial";
    var newMaterial: PhongPBMaterial= new PhongPBMaterial (
        Cast.bitmap(EarthDiffuse),
        Cast.bitmap(EarthNormal),
        CurrentPrimitive,
        Cast.bitmap(EarthSpecular)
    );
    currentPrimitive.material= newMaterial;
}
```

如图 5-26 所示为 PhongPBMaterial 材质效果。



图 5-26 PhongPBMaterial 材质效果

PhongPBMaterial 类继承了 TransformBitmapMaterial 类,这就说明,除表 5-19 所列出的这些参数外,列在 TransformBitmapMaterial 类的初始化对象参数,用于 PhongPBMaterial 类也是有效的。

表 5-19 PhongPBMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		用纹理为漫反射着色
normalMap	BitmapData		对象空间正常映射
targetModel	Mesh		应用着色的网格
specularMap	BitmapData	null	选择镜面映射 BitmapData,调整镜面反射光
gloss	Number	10	最大 Z 向着色系数
specular	Number	1	镜面反光强度

3. PhongMultiPassMaterial 材质

PhongMultiPassMaterial 类,除了可用多个外部光源照明外,其他功能与 PhongPBMaterial 类相同。

```
protected function applyPhongMultiPassMaterial():void
{
    initSphere();
    initPointLight();
    materialText.text=" PhongMultiPassMaterial";
    var newMaterial: PhongMultiPassMaterial= new PhongMultiPassMaterial (
        Cast.bitmap(EarthDiffuse),
        Cast.bitmap(EarthNormal),
        CurrentPrimitive,
        Cast.bitmap(EarthSpecular)
    );
    currentPrimitive.material= newMaterial;
}
```

PhongMultiPassMaterial 类直接继承了 TransformBitmapMaterial 类,这就说明,除表 5-20 所列出的这些参数外,列在 TransformBitmapMaterial 类的初始化对象参数,用于 PhongMultiPassMaterial 类也是有效的。

表 5-20 PhongMultiPassMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		用纹理为漫反射着色
normalMap	BitmapData		对象空间正常映射
targetModel	Mesh		应用着色的网格

续表

参数	数据类型	默认值	说 明
specularMap	BitmapData	null	选择镜面映射 BitmapData,调整镜面反射光
gloss	Number	10	最大 Z 向着色系数
specular	Number	1	镜面反光强度

4. FresnelPBMaterial 材质

FresnelPBMaterial 材质效果参考了一种现象：即看到的表面反光的总量与视角有关系。这种现象最普通的例子是往蓄水池的水面上看，如果面向水塘直接往下看，将通过水面看到下面，而没有很多反射光。但是如果从一个角度看水的表面，将会看见很多反射光线。

FresnelPBMaterial 类实现了这种效果，通过一反射球面依照各种不同的表面视角表现出四周环境的效果。

```
protected function applyFresnelPBMaterial ():void
{
    initPlane();
    materialText.text=" FresnelPBMaterial";
}
```

反射图来自球面映射，它是一个显示对象环境的，犹如从一个球面反射出的纹理图，这个纹理图嵌入到程序里，类名是 SphereMap。

```
var newMaterial: FresnelPBMaterial= new FresnelPBMaterial (
    Cast.bitmap(Water),
    Cast.bitmap(WaterNormal),
    Cast.bitmap(SphereMap),
    currentPrimitive,
    {
        smooth: true
    }
);
currentPrimitive.material= newMaterial;
}
```

在图 5-27 里，可看到 FresnelPBMaterial 的反射功能，左图的平面区反射水的蓝色材质，而当从更多的侧面角度看这个表面时，橙色的环境材质则反射出来了(如右图所示)。

FresnelPBMaterial 类直接继承了 TransformBitmapMaterial 类，这就说明，除表 5-21 所列出的这些参数外，列在 TransformBitmapMaterial 类的初始化对象参数，用于 FresnelPBMaterial 类也是有效的。

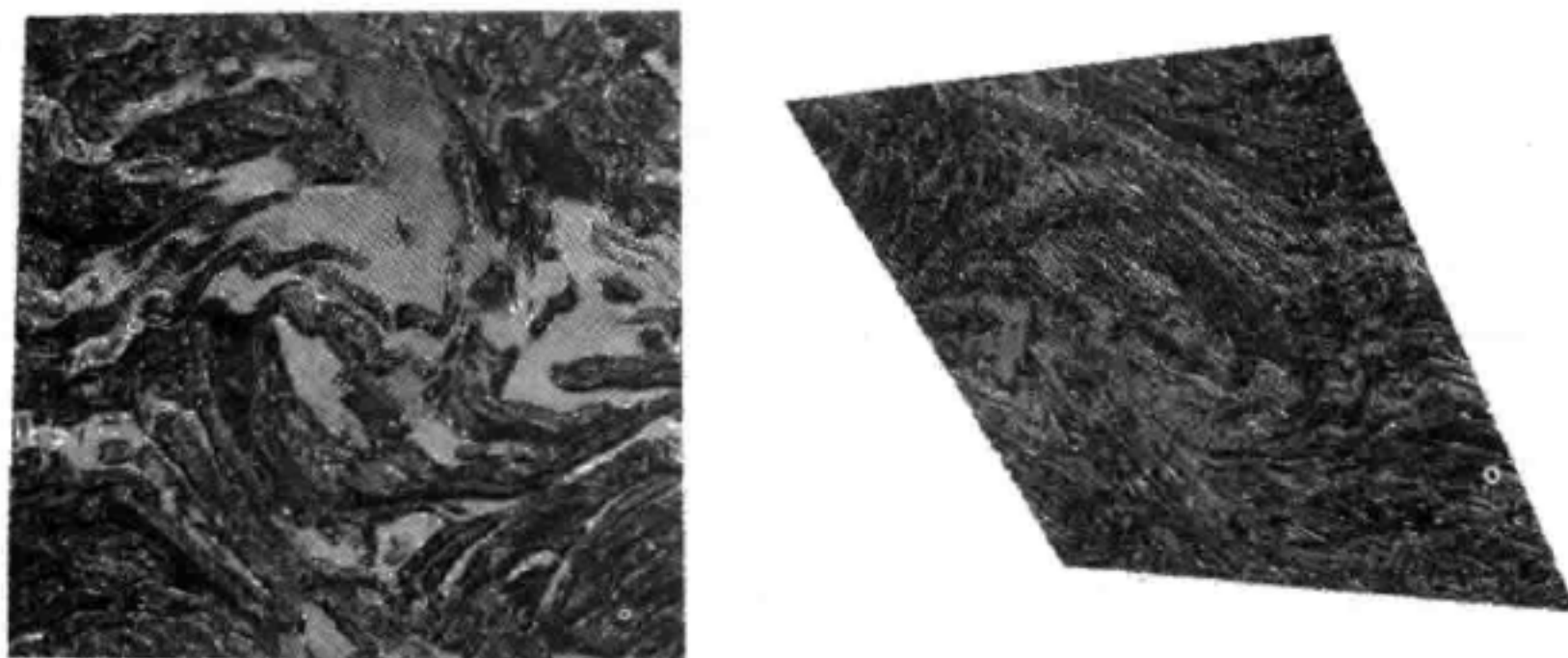


图 5-27 FresnelPBMaterial 材质效果

表 5-21 FresnelPBMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		用纹理为漫反射着色
normalMap	BitmapData		对象空间正常映射
faces	Array		大小相等的正方形纹理数组,用于立方体每个面
targetModel	Mesh		应用着色的网格
envMapAlpha	Number	1	不透明的环境映射,即如何反射表面,1 是镜面
outerRefraction	Number	1.0008	折射四周的索引
innerRefraction	Number	1.330	折射材质的索引
refractionStrength	Number	1	折射漫射纹理上的最大数,用于水类的材质

5. CubicEnvMapPBMaterial 材质

立方体环境映射 PB 材质 CubicEnvMapPBMaterial 使用环境映射,立方体各面上的反射纹理添加到基于法向贴图的纹理,应用到 3D 对象。

```
protected function applyCubicEnvMapPBMaterial():void
{
    initSphere();
    materialText.text=" CubicEnvMapPBMaterial";
}
```

这些反射来自立方体的 6 个纹理图,这些纹理图显示 3D 对象四周的环境样子,6 个纹理图放置在 sky Array 数组里。

```
var sky:Array=new Array();
sky[CubeFaces.LEFT]=Cast.bitmap(skyleft);
sky[CubeFaces.FRONT]=Cast.bitmap(skyfront);
sky[CubeFaces.RIGHT]=Cast.bitmap(skyright);
```

```
sky[CubeFaces.BACK] = Cast.bitmap(skyback);
sky[CubeFaces.TOP] = Cast.bitmap(skyup);
sky[CubeFaces.BOTTOM] = Cast.bitmap(skydown);
var newMaterial: CubicEnvMapPBMaterial = new CubicEnvMapPBMaterial (
    Cast.bitmap(EarthDiffuse),
    Cast.bitmap(EarthNormal),
    sky,
    currentPrimitive,
    {
        envMapAlpha: 0.5
    }
);
currentPrimitive.material = newMaterial;
}
```

如图 5-28 所示为 FresnelPBMaterial 材质效果。

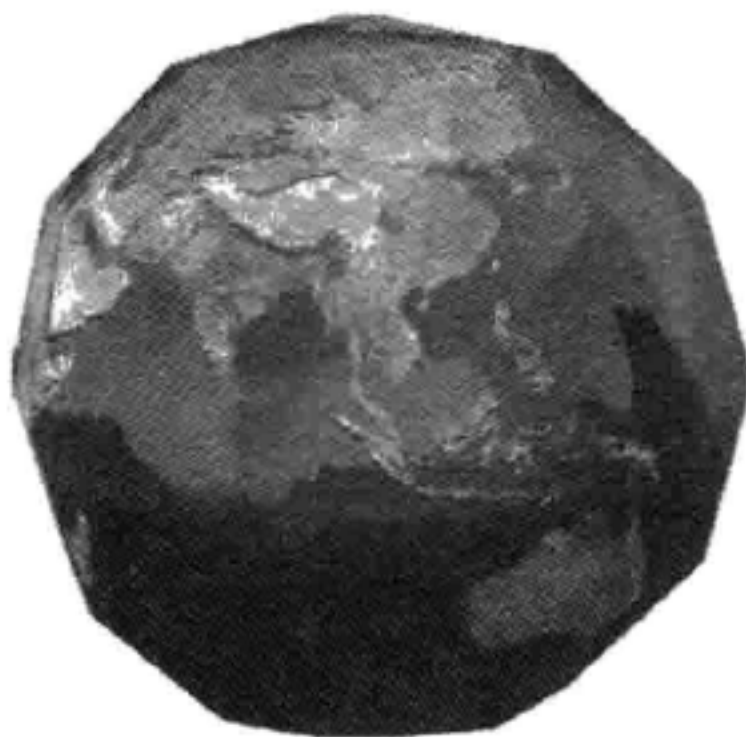


图 5-28 FresnelPBMaterial 材质效果

CubicEnvMapPBMaterial 类继承了 TransformBitmapMaterial 类,这就说明,除表 5-22 所列出的这些参数外,列在 TransformBitmapMaterial 类的初始化对象参数,用于 CubicEnvMapPBMaterial 类也是有效的。

表 5-22 CubicEnvMapPBMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
bitmap	BitmapData		用纹理为漫反射着色
normalMap	BitmapData		对象空间正常映射
faces	Array		大小相等的正方形纹理数组,用于立方体每个面
targetModel	Mesh		应用着色的网格
envMapAlpha	Number	1	不透明的环境映射,即如何反射表面,1 是镜面

5.8.9 从外部文件载入纹理图

把资源嵌入到 SWF 文件,虽然是经常用且十分方便的方法,但有些时候它并不是令人满意的。Away3D 提供了很多类,它们帮助我们载入外部资源。

当访问外部资源的时候,有些要考虑的事情,其中之一是载入的进程是异步的,这就表明实际下载外部资源的进程是在后台完成的。正常情况下,当下载进程完成调用 Event.COMPLETE 处理事件,它又调用回调函数的方法一次性地恢复数据,这些数据就是所需要的资源。

对于一个简单的 Bitmap 材质,BitmapFileMaterial 类处理后台载入的过程,但要提供下载的外部纹理图像的网址 URL 和管理所有异步下载进程的 BitmapFileMaterial 类。

其他的材质类型,没有等效的 BitmapFileMaterial 类管理载入外部纹理的过程,对这些 Away3D 提供了 TextureLoadQueue 类,它能将很多纹理资源作为一组载入,并在资源准备好的时候通知用户。

1. Bitmap 文件材质

正如下面的代码所示,BitmapFileMaterial 类的使用非常简单,它隐藏了异步载入过程的细节,使 BitmapFileMaterial 类就像使用其他材质类一样简单。所要做的所有工作,只是提供要载入的纹理的地址。

```
protected function applyBitmapFileMaterial():void
{
    initSphere();
    materialText.text="BitmapFileMaterial";
    var newMaterial: BitmapFileMaterial= new BitmapFileMaterial ("earth_diffuse.jpg");
    currentPrimitive.material= newMaterial;
}
```

BitmapFileMaterial 类继承了 BitmapMaterial 类,这就说明,除表 5-23 所列出的这些参数外,列在 BitmapMaterial 类的初始化对象参数,用于 BitmapFileMaterial 类也是有效的。

表 5-23 BitmapFileMaterial 类构造函数的参数

参数	数据类型	默认值	说 明
url	String		定义载入纹理图像的地址
checkPolicyFile	Boolean	false	指定用于载入文件的 LoaderContext 对象的 checkPolicyFile 的属性值

2. 使用纹理载入队列

如果必须载入若干个纹理图来建立一个新的材质,这个过程会变得稍为复杂一些,以

Dot3BitmapMaterial 类作为例子,它既要纹理贴图也要法向贴图。

TextureLoadQueue 能够用一个组载入多个外部资源,可以给 Dot3BitmapMaterial 类提供初始参数载入全部需要的纹理,当全部纹理载入后建立这个类的新实例。

```
protected function applyExtenalDot3BitmapMaterial():void
{
    initSphere();
    initDirectionalLight();
    materialText.text="Extenal Dot3BitmapMaterial";
}
```

首先,建立一个 TextureLoadQueue 类的实例:

```
var textureLoadQueue:TextureLoadQueue=new TextureLoadQueue();
```

然后,必须对每一个要载入文件建立两个附加的对象,第一个对象是 URLRequest,它的构造函数的参数取的是第一个外部文件的网址 URL。

```
var req:URLRequest=new URLRequest("earth_diffuse.jpg");
```

第二个对象是 TextureLoader 对象:

```
var loader:TextureLoader=new TextureLoader();
```

然后,把这两个对象都传递到 TextureLoadQueue 类的 addItem()函数中:

```
textureLoadQueue.addItem(loader, req);
```

对所要载入的其他各个纹理文件,重复上述过程:

```
req:URLRequest=new URLRequest("earth_normal.jpg");
loader:TextureLoader=new TextureLoader();
textureLoadQueue.addItem(loader, req);
```

当 TextureLoaderQueue 对象一旦完成全部外部纹理的下载过程时,它就调用 Event.COMPLETE 事件处理函数。一旦这个事件被调用,就能访问所请求的 Bitmap 数据建立材质。为方便起见,先建立一个响应此事件的无名函数。

```
textureLoadQueue.addEventListener(
    Event.COMPLETE,
    function(event:Event):void
    {
```

建立两个 BitmapData 变量,把它们赋值给刚载入的两个外部文件的数据容器:

```
var diffuse:BitmapData;
var normal:BitmapData;
```

但 TextureLoaderQueue 并没有提供一个容易访问的方法用于检索出载入的 images,

代替的是它提供了一个 TextureLoader 对象数组,数组的各个元素能够识别和处理载入的各个 images。这就要用循环的方法,在全部数组里找出相应外部文件的 TextureLoader 对象。

```
For each(var image:TextureLoader in textureLoadQueue.image)
{
```

不管当前获得的 TextureLoader 对象里的数据是从哪个外部文件来的,在循环体内,首先新建一个 BitmapData 对象,把每次循环中的 TextureLoader 对象的内容画到 BitmapData 中。

```
var bitmapData:BitmapData=new :BitmapData(image.width,image.height);
bitmapData.draw(image);
```

使用当前的 TextureLoader 对象的文件名属性,能找出与它对应的外部文件名,这可使我们引用新建的带有 diffuse 和 normal 变量名的 BitmapData 对象。

```
If(image.filename=="earth_diffuse.jpg")
    diffuse=bitmapData;
else If(image.filename=="earth_normal.jpg")
    normal=bitmapData;
}
```

现在有了可用的纹理图和法向贴图的 Bitmap,把它们直接用于建立 Dot3BitmapMaterial 材质对象。

```
currentPrimitive.material=new Dot3BitmapMaterial(diffuse,normal);
}
};
```

最后,在适当的地方,通过调用带有匿名的回调函数的 TextureLoaderQueue 的 start() 函数,让它开始载入这些文件。

```
textureLoadQueue.start();
}
}
}
```

使用 TextureLoaderQueue 类载入两个外部文件的时候,Flash 或 Flex 有相同逻辑的功能载入一个或十几个文件的类,也可以用 Flash 或 Flex 的载入器类 Loader 直接载入外部资源。

模型和动画

正如在第 2 章中所介绍到的,从最基础开始使用基本类元素,如顶点、三角形面、Sprite3D 对象和段,组建一个 3D 对象是可能的。然而,对于更复杂的模型,手工编码每个元素却是不切实际的。虽然 Away3d.primitives 包中的很多类提供了解决这一问题的办法,规定了一个快速建立某些标准模型的方式,但是一些高级的应用程序必须显示更加复杂的模型,在这种情形下,这些 Away3d 标准原始模型并没有提供足够的灵活性。幸好 Away3d 能够装入并显示由外部建模应用软件建立的 3D 模型。3D 建模应用软件是特别设计的一套软件,它提供了一个可视化的环境,在可视的环境里,可以创建、操作、控制和修改 3D 模型。用建模应用软件创建和编辑 3D 网格,肯定比用 ActionScript 编码建立网格要方便很多。

Away3D 能直接装入范围很广的各种 3D 格式模型,把 3D 网格输出到模型文件里,这些文件又能为 Away3D 所使用。能这样处理的建模应用软件包括如下几个。

(1) 3ds Max:一个流行的商业化的用于建模、动画和渲染的建模应用软件,运行在 Windows 上。

(2) Blender:一个免费的开源的建模应用软件,能运行在很多平台上,如 Windows、Linux 和 MacOS。

(3) Milkshape:一个商业化的低平面的模式化应用软件,运行在 Windows 上,主要面对 Half-Life(CS)游戏设计。

(4) Sketch-up:一个免费的 3D 建模应用软件,由 Google 提供。它的商用版也是可用的,包含很多附加特性。Sketch-up 运行在 Windows 和 MacOS 上。

实际如何使用这些 3D 建模应用软件,已经超出了本书的范围,只要了解所提供的 3D 模型对象,装入它们,把它们输出到文件,允许运行这些过程,就没有必要知道是如何从头开始建立 3D 模型的。

本章主要内容:

- 从很多 3D 建模应用软件输出模型文件。
- 在 Away3D 中,既可从嵌入资源文件里,也可从外部文件里,载入模型文件。
- 将模型文件转换成 ActionScript 的 AS 类。

6.1 Away3D 支持的 3D 格式

Away3D 包含很多类,它们能装入范围很广的 3D 模型文件格式,全都支持静态 3D 模型的格式的装入和运行,而只有很少的能装入动画模型。表 6-1 列出了 Away3D 支持的模型格式、格式的扩展名、格式能否装入动画模型,以及能装入并能解析它们的 Away3D 类。

表 6-1 Away3D 支持的模型格式

格式	扩展名	是否支持静态模型	是否支持动画模型	Away3D 类
Collada	DAE	*	*	Collada
Quake II	MD2	*	*	MD2
3ds Max Ascii	ASE	*		Ase
Away3D	AWD	*		AWData
Google Earth	KMZ	*		Kmz
3ds Max	3DS	*		Max3DS
Wavefront	OBJ	*		Obj
ActionScript	AS	*	*	

6.2 输出 3D 模型

下面介绍如何从很多不同的 3D 建模应用程序里,输出一个 Collada 文件。Collada 文件是开放的,基于 XML 格式的,设计这种 XML 格式就是为了在不同的 3D 应用之间,提供一种交换数据的方法。Away3D 既支持从 Collada 文件里装入静态模型,也支持从 Collada 文件里装入动画模型。

6.2.1 从 3ds Max 输出模型文件

3ds Max 是一个商业化的建模应用软件,在编写本书时,ColladaMax 插件的最新版本

是 3.05C,是一个用于输出 3D 模型的插件,这个版本支持 3ds Max 2008,3ds Max 9,3ds Max 8 SP3 和 3ds Max 7 SP1。注意这个版本不支持 3ds Max 2010 和 3ds Max 2011。

有个 3ds Max 9 的试用版可以使用,虽然很难找到,但如果在因特网上搜索 Autodesk3dsMax2009_ENU_TrialDownload.exe,可找到它的一个程序,上述 exe 执行文件是 3ds Max 9 试用版的安装文件名。

(1) 从 <http://sourceforge.net/projects/Colladamaya/files/> 下载并安装 ColladaMax 插件;

(2) 打开 3ds Max;

(3) 单击左上角的文件,选择希望打开的文件,单击 Open;

(4) 单击左上角的文件,选择 Export;

(5) 从选择保存文件类型的下拉列表框中,选择 Autodesk Collada(*.DAE);

(6) 选择与原 Max 文件保存位置相同的目录;

(7) 在 File name 输入框内,输入要保存的输出文件的名称,单击 Save 按钮,输出文件;

(8) 在如图 6-1 所示的 ColladaMax Export 对话框里,确保以下几个复选框已勾选:

① Relative Paths;

② Normals;

③ Triangulate;

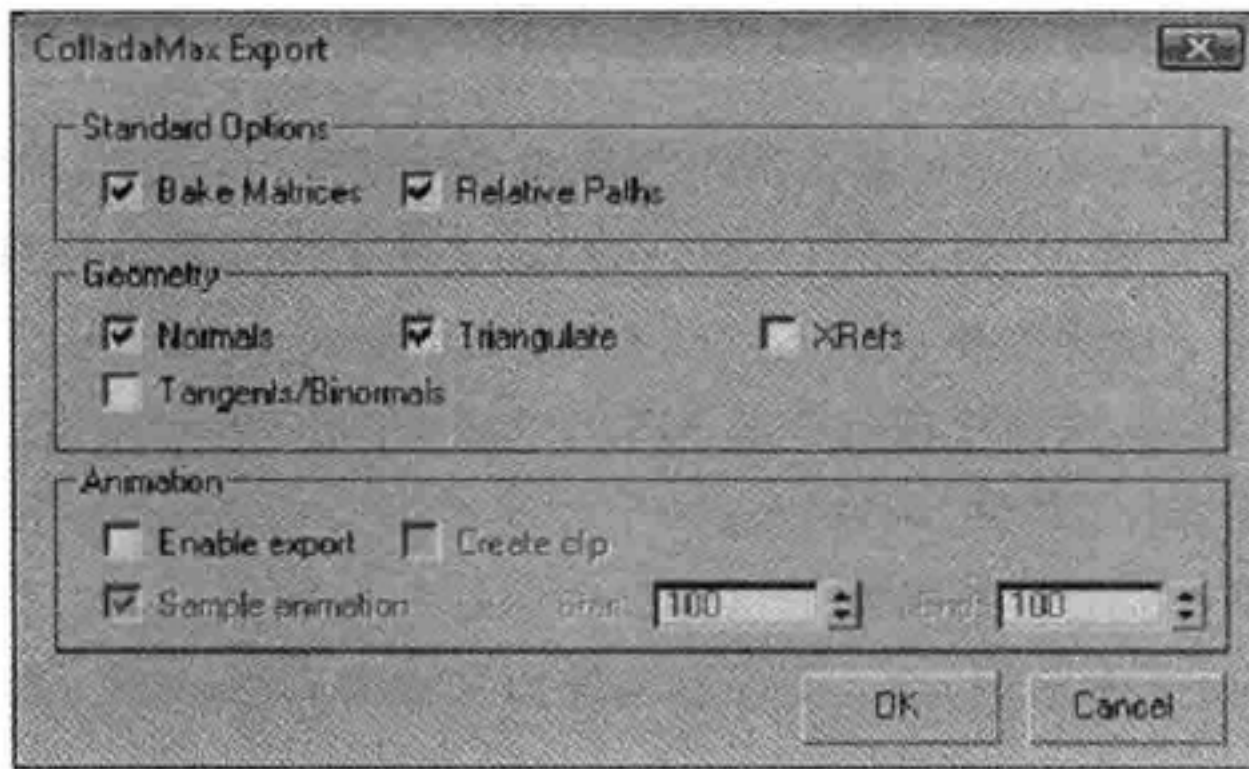


图 6-1 ColladaMax Export 对话框

(9) 如果想输出动画,勾选 Enable export 复选框;

(10) 如果想输出一个特殊的帧范围,勾选 Sample animation 复选框,并且在 Start 和 End 文本框里,输入请求的值;

(11) 单击 OK 按钮输出此文件,如图 6-2 所示。



由 MilkShape 提供的 Collada 输出器不支持动画模型输出,于是,即使载入包含动画模型的 MilkShape MS3 文件,它输出的 Collada DAE 文件也将是静态的网格。MilkShape 的试用版可从它的网站下载并安装: <http://chumbalum.swissquake.ch>。

- (1) 单击 File|Open, 选择希望打开的 MS3D 文件, 单击 Open 按钮;
- (2) 单击 File|Export|COLLADA…;
- (3) 选择保存原 MS3D 文件位置的目录;
- (4) 在 File name 文本框内, 输入要保存的输出文件的名称, 单击 Save 按钮。

同 MilkShape 一样, Sketch-up 也不支持动画的 Collada 模型文件输出, Sketch-up 能从 Google 网站免费下载: [http:// Sketchup. google. com/](http://Sketchup.google.com/)。

- (1) 单击 File|Open 命令,选择希望打开的 SKF 文件,单击 Open 按钮;
- (2) 单击 File|Export|3D Model…;
- (3) 从 Export Type 组合框里,选择 Collada File(*.dae);

- (4) 选择相应的目录,在 File name 文本框内,输入要保存的输出文件的名字;
- (5) 单击 Options...按钮;
- (6) 在图 6-3 中,确保勾选 Triangulate All Faces 复选框;
- (7) 如果勾选 Export Texture Maps 复选框,Sketch-up 将同 DAE 文件一起也输出纹理图;
- (8) 单击 OK 按钮,保存选项;
- (9) 单击 Export 按钮,输出文件。

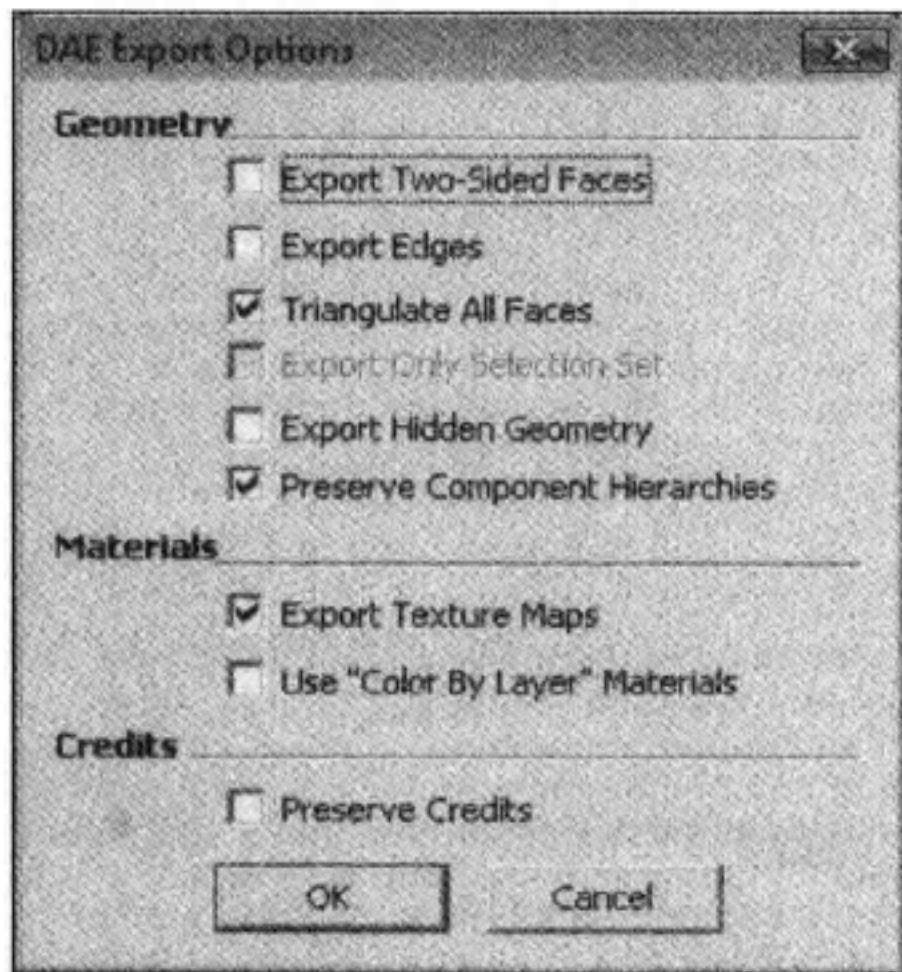


图 6-3 DAE 参数设置对话框

6.2.4 从 Blender 输出模型

编写这本书时,Blender Collada 输出器的最新版是 0.3.162,支持输出动画,然而在大多数情况下,Away3D 并不能正确载入这些动画模型。推荐从 Blender 仅把静态网格载入到 Collada 文件中。

- (1) 单击 File|Open...命令选择希望打开的 BLEND 文件,单击 Open 按钮;
- (2) 单击 File|Export|COLLADA1.4(*.dae)...;
- (3) 在存放原 BLEND 文件位置的目录里,在 Export File 文本框内,输入要保存的输出文件的名字;
- (4) 确保 Triangles 和 Use Relative Path 按钮按下,如图 6-4 所示;
- (5) 单击 Export and Close 按钮。

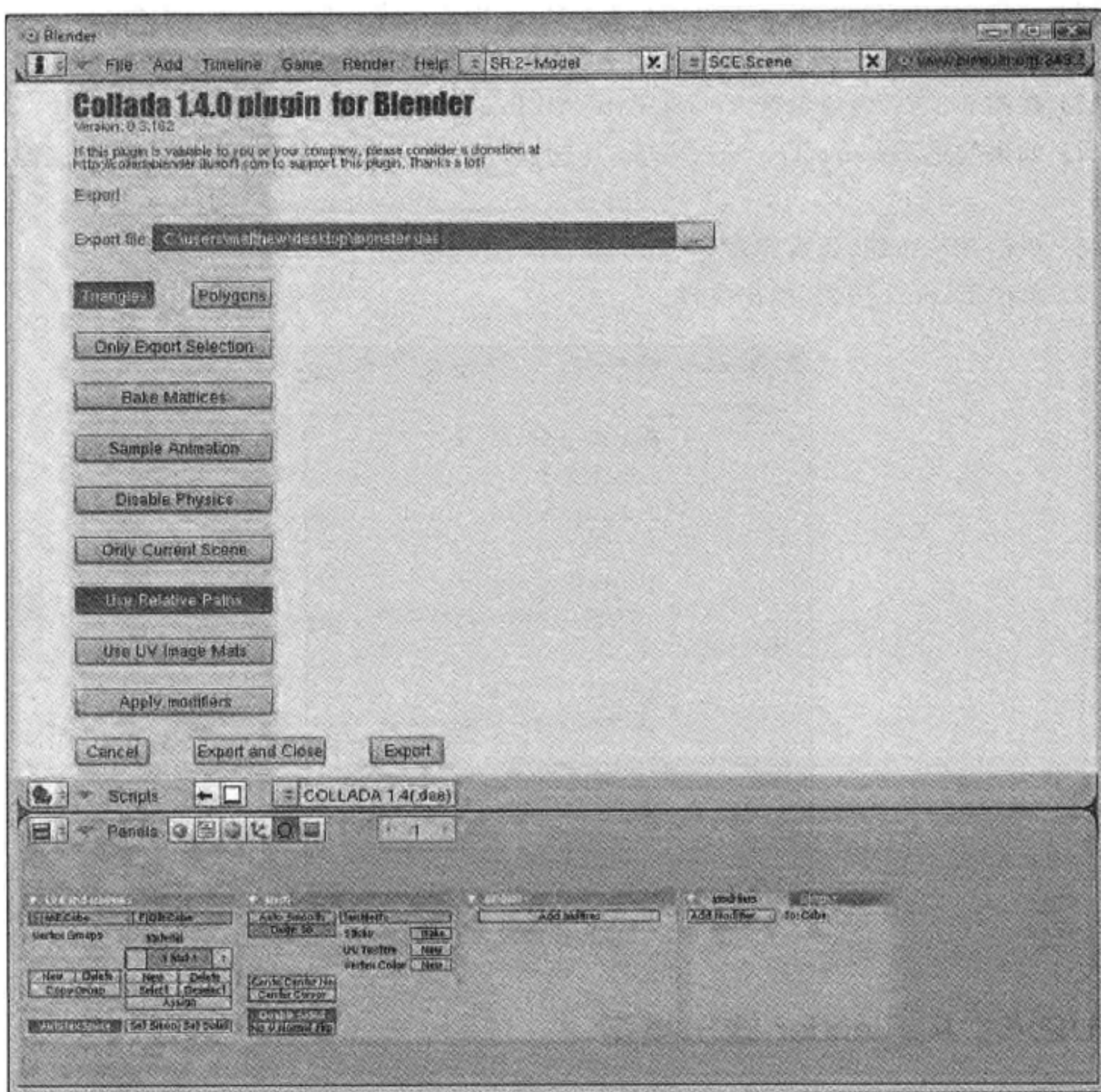


图 6-4 Blender 输出模型窗口

6.2.5 有关 Collada 输出器的注意事项

尽管 Collada 是免费的标准的开源软件,其输出的 Collada 模型文件,能被 Away3D 正确地解读是碰巧的事,3ds Max 里的 Collada 输出器就是一个最好的例子,在测试的时候,不论是 3ds Max 里内置的 Collada 输出器,还是第三方的 OpenCollada 输出器(<http://opencollada.org/>,在编写本书时,最新版是 1.2.5)输出 Away3D 能读入的动画的 Collada 文件,即使在最好的 Away3D 里它显示的只是个静止的网格。最差的情况下,当 Away3D 读入 DAE 文件时会抛出一个异常信息。Blender 带的 Collada 输出器也是一样,虽然它声

称能输出与 Away3D 兼容的动画 Collada 网格,但实际上也是个静止的网格。

了解这一点是重要的,因为对于 3D 建模应用软件来说,能输出 Collada 文件,但并不能保证该文件能被 Away3D 正确读入。

6.3 载入 3D 模型

载入一个嵌入式 3D 模型的一般步骤如下。

- (1) 导入一些必要的类;
- (2) 建立一个继承 Away3DTemplate 的类;
- (3) 嵌入模型文件和纹理文件;
- (4) 建立一个调用 Away3DTemplate 构造函数的构造函数;
- (5) 重载 initScene() 函数:
 - ① 从嵌入的纹理资源建立材质;
 - ② 使用载入类模型的 parse() 函数,建立 Object3D、Mesh 和 ObjectContainer3D 对象;
 - ③ 把材质赋给载入的 3D 对象;
 - ④ 把 3D 对象加入到场景;
 - ⑤ 启动支持动画模型的动画。

载入一个外部 3D 模型文件的一般步骤与上面的步骤相似。

- (1) 导入一些必要的类;
- (2) 建立一个继承 Away3DTemplate 的类;
- (3) 建立一个调用 Away3DTemplate 类构造函数的构造函数;
- (4) 重载 initScene() 函数:
 - ① 使用载入类模型的 Load() 函数,建立载入的 3D 对象;
 - ② 把载入 3D 对象加入到场景;
 - ③ 指派响应 Loader3DEvent. LOAD_SUCCESS 事件的函数;
 - 如果必要的话,手工应用材质;
 - 启动支持动画模型的动画。

6.4 动画模型

Collada 的 DAE, Quake2 的 MD2 和 ActionScript 的 AS 模型,它们都是能用于载入动画 3D 对象的一些格式,但是用于载入各个动画类的时候,彼此间每个格式都有些微小的区

别,特别是在载入嵌入资源或载入它们的外部文件的选项上。

6.4.1 MD2 载入一个嵌入式文件

MD2 是 Quake2 所使用的模型格式,这些模型是 Away3D 使用的理想格式,因为它们具有较低平面计算并支持动画。下面建立一个 MD2EmbeddedDemo 的应用程序,来展示嵌入 Md2 和载入 MD2 文件的方法。

```
package  
{
```

被解析的 3D 对象将作为一个网格 Mesh 返回。

```
public class MD2EmbeddedDemo extends Away3DTemplate  
{  
import away3d.core.base.Mesh;
```

使用由 Cast 类提供的静态函数,来投射不同类型间的对象。

```
import away3d.core.utils.Cast;
```

下面输入用来载入 MD2 文件的这些类:

```
import away3d.loaders.Md2;
```

把 BitmapMaterial 材质应用到 3D 对象,参考第 5 章,材质重载 BitmapMaterial 材质的细节。

```
import away3d.materials.BitmapMaterial;
```

AnimationData 类包含可以一次性载入并启动绘画 3D 对象的函数。

```
import away3d.loaders.data.AnimationData;
```

MD2 模型能够嵌入,但是因为 ActionScript 编译器不理解 MD2 格式,所以要把它们作为原始数据文件嵌入(即 application/octet-stream 的 MIME 类型)。

```
[Embed(source="ogre.md2", mimeType="application/octet-stream")]  
protected var MD2Model:Class;
```

默认地,对 MD2 的模型纹理用的是 PCX 格式,它不被 Away3D 支持。在这里已经把原 PCX 图像 image 文件格式变换成 JPG image 文件格式,然后再嵌入,所以不必指定 MIME 类型。因为 ActionScript 编译器理解 JPG image 图像文件格式。

```
[Embed(source="ogre.jpg")]  
protected var MD2Material:Class;
```

代表 3D 对象的 Mesh 类,通过 md2Mesh 属性来引用:

```
protected var md2Mesh:Mesh;
```

构造函数调用基类 Away3DTemplate 的构造函数,它初始化 Away3D 引擎。

```
public function MD2EmbeddedDemo()
{
    super();
}
```

对载入 MD2 文件重载 initScene()函数,把产生的 3D 对象添加到场景。

```
protected override function initScene():void
{
    super.initScene();
```

首先建立一个新的 BitmapMaterial 对象,表示嵌入的 image 图像文件。

```
var modelMaterial:BitmapMaterial=new BitmapMaterial(Cast.bitmap(MD2Material));
```

使用 Md2 类的解析函数 parse()载入嵌入的 MD2 文件,用 Cast 类的 bytearray()函数将 MD2 文件转换成字节数组,解析函数 parse()返回网格 Mesh 对象,然后将网格 Mesh 对象赋给 md2Mesh 属性。

```
md2Mesh=Md2.parse(Cast.bytearray(MD2Model),
{
```

对场景里的 3D 对象的缩放、位置和旋转指定必要的初始化参数,使它在屏幕上显示得漂亮些,在第 3 章里,学习了更多的这样的转换 3D 对象的方法。

```
    scale: 0.01,
    z: 100,
    rotationY: -90
});
```

然后,通过 material 属性给 3D 对象指定材质。

```
md2Mesh.material=modelMaterial;
```

不像原先使用的原始 3D 对象,通过 material 初始对象参数指定材质,这种方法不能用于 MD2 类载入的 3D 对象。以下的初始化和初始对象的问题这一节,对此将有详细的解释。

把 3D 对象添加到场景,使它可见:

```
scene.addChild(md2Mesh);
```

大多数的 MD2 模型中定义了很多动画,如 stand、run、attack 和 jump,这些动画的名字与游戏里的角色的动作一致,Quake 2 里虽然这些动画的名字是普通常用的,但不能保证这些动画的名字包含在 MD2 模型文件里,在启动所希望的动画之前,首先要检查希望的动画是否包含在载入的模型对象中。

```
var animationData: AnimationData = md2Mesh.animationLibrary.getAnimation("stand");
```

如果 animationData 变量不空,则说明载入的 3D 对象包含希望启动的动画。

```
If(animationData != null)
```

动画能够通过调用 play() 函数启动:

```
animationData. animator. play();
}
```

6.4.2 MD2 载入一个外部文件

载入一个外部的 MD2 文件步骤,与载入一个嵌入式的 MD2 文件步骤非常相似,下面建立一个 MD2ExternalDemo 的应用程序,来载入并显示外部 MD2 文件,并看它与上面的 MD2EmbeddedDemo 有什么不同。

```
package
{
import away3d.core.base.Mesh;
```

需要注册一个调用函数,以便当 3D 对象载入后能够启动初始化动画,这个函数将把 Loader3DEvent 对象当作一个参数:

```
import away3d.events.Loader3DEvent;
```

这时 Md2 类返回的不是 Mesh 对象,而是一个 Loader3D 对象,当 3D 对象载入的时候,Loader3D 对象被用来作为一个占位符。

```
import away3d.loaders.Loader3D;
import away3d.loaders.Md2;
import away3d.loaders.data.AnimationData;
```

BitmapFileMaterial 类提供了一个方便的方法,来载入外部 image 文件,并把它作为材质,第 5 章中介绍了 BitmapFileMaterial 类的详细信息。

```
import away3d.materials.BitmapFileMaterial;
```



```
public class MD2ExternalDemo extends Away3DTemplate
{
    protected var mesh:Mesh;
    public function MD2ExternalDemo()
    {
        super();
    }
    protected override function initScene():void
    {
        super.initScene();
    }
}
```

当载入一个外部文件的时候,调用 Md2 类的 load() 函数,第一个参数是 MD2 文件的网站 URL。

由于 loader3D LoadTextures() 函数的 bug 原因,对 load() 函数提供的 URL 网址,需要以一个斜杠开头,即使这个文件与 SWF 文件在同一个文件夹下。

load() 函数将返回一个 Loader3D 对象,Loader3D 对象是一个当 3D 对象载入时用于显示的占位符。

```
var placeholder:Loader3D=Md2.load("./ogre.md2",
{
    scale: 0.01,
    z: 100,
    rotationY: -90
});
```

当 Loader3D.LOAD_SUCCESS 事件被调用时,3D 对象已经载入并解析,而且已做好被使用的准备,这时,希望设置初始动画,于是注册一个在 Loader3D.LOAD_SUCCESS 事件触发时被调用的函数 onLoadSuccess()。

```
placeholder.addEventListener(
    Loader3DEvent.LOAD_SUCCESS,
    onLoadSuccess);
```

把占位符的 Loader3D 对象加入到场景,当 3D 对象被载入时,3D 对象加到场景,占位符的 3D 对象(它是 Cube 原始模型)就被移除。

```
scene.addChild(placeholder);
}
protected function onLoadSuccess(event:Loader3DEvent):void
{
```

使用 onLoadSuccess() 函数可以引用载入的 3D 对象。

```
mesh=event.loader.handle as Mesh;
```

Md2 类能够通过 MD2 文件里的纹理信息建立自己的材质,由于 Flash 不支持 PCX 的格式,而 PCX 格式是 MD2 模型默认的格式,Flash 试图载入名字与 MD2 文件里引用的 PCX 格式文件名相同的 image 文件。Flash 所用的这一文件新的扩展名,能够由默认的 JPG 改为另一个 Flash 支持的扩展名,如 PNG 或 GIF,只要指定 pcxConvert 的初始化对象参数就行,该参数是 Md2 的 load() 函数支持的。

然而,通过 MD2 文件引用纹理文件时常是错误的,或包含很长的路径,如 quake2/baseq2/players/modelname/texture.pcx。这种不可预测的纹理文件名最好避免。为此,建立一个新的 BitmapFileMaterial 实例,把纹理文件的网址 URL 传递给它的构造函数,在材质使用到载入 3D 对象时,通过 material 属性来指定。

```
mesh.material=new BitmapFileMaterial("ogre.jpg");
```

然后调用 stand 启动动画

```
var animationData:AnimationData=mesh.animationLibrary.getAnimation("stand");
If(animationData!=null)
    animationData.animator.play();
}
```

6.4.3 Collada 载入一个嵌入式文件

载入一个嵌入式的 Collada 模型文件的步骤,与载入一个嵌入式的 MD2 文件的步骤相似:模型文件和纹理都是嵌入的,并且 3D 对象的建立使用载人类模型(在这种情形下,名字为 Collada)的 parse() 函数。

```
package
{
    import away3d.containers.ObjectContainer3D;
    import away3d.core.utils.Cast;
```

Collada 类用于解析嵌入的 Collada DAE 文件

```
import away3d.loaders.Collada;
import away3d.loaders.data.AnimationData;
import away3d.materials.BitmapMaterial;
import away3d.events.Event;
public class ColladaEmbeddedDemo extends Away3DTemplate
{
```

嵌入的 Collada DAE 文件如同原始数据文件一样,因为 Collada DAE 文件实际上就是一个 XML 文件,所以可以指定其 MIMETYPE 的类型是“text/xml”,

```
[Embed(source="beast.dae", mimeType="application/octet-stream")]
protected var ColladaModel:Class;
[Embed(source="beast.jpg")] protected var ColladaMaterial:Class;
public function ColladaEmbeddedDemo()
{
    super();
}
protected override function initScene():void
{
    super.initScene();
    var modelMaterial:BitmapMaterial=new
        BitmapMaterial(Cast.bitmap(ColladaMaterial));
```

在这里,使用 Collada 类里的静态函数 parse() 建立 ObjectContainer3D 对象,容纳模型文件里的一些网格和动画:

```
var colladaContainer:ObjectContainer3D=
    Collada.parse(Cast.bytearray(ColladaModel),
```

在 Collada 文件的模型中,能使用很多分开的材质完成最后的显示。这个例子使用的 Collada 文件中仅用了一种材质,对于多种材质,编程思维的逻辑是相同的。定义一个叫做 material 的初始对象参数,而且在该初始对象参数里,指派了另一个初始对象,它把 Away3D 的材质映射到 Collada 文件里定义的材质名字。在这个例子中,定义在 Collada 文件里的单一材质叫做 monster。

```
{
    materials:
    {
        monster:modelMaterial
    },
}
```

然后,旋转 3D 对象,以便它在屏幕上显示得漂亮些。

```
rotationY: 90
}
);
```

增加 3D 对象的缩放功能,使它在屏幕上容易看见:

```
colladaContainer.scaleX=
colladaContainer.scaleY=
colladaContainer.scaleZ=20;
scene.addChild(colladaContainer);
```

这里,得到一个引用 AnimationData 的对象,它保存一个叫做 default 的动画。

```
var animationData: AnimationData = colladaContainer.  
    animationLibrary.getAnimation("default");
```

如果动画存在,然后启动它。

```
    If(animationData != null)  
        animationData.animator.play();  
    }  
}  
}
```

6.4.4 Collada 载入一个外部文件

载入一个外部 Collada 文件,与载入一个嵌入式 Collada 文件相似,但有很大的不同:

①不必手工分配材质;②载入模型完成的事件被调用时,动画就启动。

```
package  
{
```

载入一个外部文件,是个异步的过程,Loader3DEvent 类被注册到响应 Loader3DEvent.LOAD_SUCCESS 事件的处理函数,这样可以知道文件是否载入成功。

```
import away3d.events.Loader3DEvent;  
import away3d.loaders.Collada;
```

当 Collada 文件正在载入时,Loader3D 类被用作为一个占位符。

```
import away3d.loaders.Loader3D;  
import away3d.loaders.data.AnimationData;  
import flash.events.Event;  
public class ColladaExternalDemo extends Away3DTemplate  
{  
    public function ColladaExternalDemo()  
    {  
        super();  
    }  
    protected override function initScene():void  
    {  
        super.initScene();  
    }  
}
```

这里使用了 Collada 类的静态函数 load(),这个函数需要载入的 Collada 文件的网址 URL(记住,在网址前加斜杠,即使 Collada 文件与 SWF 文件在同一个文件夹)并返回 Loader3D 对象。不必提供有关使用材质的信息,通过 DAE 文件里引用下载的 image 文件,Collada 类即可以建立材质。

```
var placeHolder:Loader3D = Collada.load("./beast.dae",
```



```

    {
        rotationY: 90
    }
);

```

当调用处理 Loader3DEvent. LOAD_SUCCESS 事件的响应函数时, Loader3D 类的 addOnSuccess() 函数提供了一快捷方式注册被调用的函数:

```
placeholder.addOnSuccess(onLoadSuccess);
```

把 Loader3D 对象加入到场景, 在 Collada 文件正在载入的时候, 它显示原始的立方体:

```
Scene.addChild(placeholder);
}
```

当 onLoadSuccess 函数被调用时, 能缩放 3D 对象, 得到访问默认动画的数据, 然后如果动画存在就启动它。

```

protected function onLoadSuccess(event: Loader3DEvent): void
{
    event.loader.handle.scaleX =
    event.loader.handle.scaleY =
    event.loader.handle.scaleZ = 20;
    var animationData: AnimationData =
        event.loader.handle.animationLibrary.getAnimation("default");
    if(animationData != null)
        animationData.animator.play();
}
}

```

6.4.5 AS 载入转换后的模型

模型也能用 ActionScript 类来定义, 回忆第 2 章中的海龟原始模型, 它是一个复杂的模型, 能通过实例化 SeaTurtle 类来建立。

Collada DAE 和 Quake 2 MD2 的模型格式, 前面已经证明两者都能从外部文件和嵌入式资源载入。因为 ActionScript 类的特点, 从外部文件载入 ActionScript 模型是不可能的, 这就是为什么仅用一个应用程序, 来展示使用存储在 AS 文件中模型的方法。

对于这个应用程序, 使用一个叫做 Ogre 的类, 它是由 MD2 模型转换而来的, MD2 模型就是使用上面的 MD2ExternalDemo 和 MD2EmbeddedDemo 的类, 建立一个像 Ogre 类过程的详细介绍, 请参见把载入的模型转换成 ActionScript 类一节。

```

package
{
    import away3d.core.base.Mesh;

```

```
import away3d.core.utils.Cast;
import away3d.materials.BitmapMaterial;
public class AS3ModelDemo extends Away3DTemplate
{
    [Embed(source="ogre.jpg")]
    protected var AS3Material:Class;
    protected var model:Mesh;
    public function AS3ModelDemo()
    {
        super();
    }
    protected override function initScene():void
    {
        super.initScene();
        var modelMaterial:BitmapMaterial=
            new BitmapMaterial(Cast.bitmap(AS3Material));
```

正像一个原始 3D 对象一样,Ogre 模型的建立是通过实例化一个标准的 ActionScript 类,不必像 Collada 或 Md2 那样使用载入和解析模型格式文件的中间类。

在下面的代码段里,读者可能已经注意到,我们已传递了一个初始化对象参数 scaling,然后通过网格 Mesh 类的属性直接设置了模型 3D 对象的材质和位置。这是因为用于建立 AS 特定类的这一工具,它仅读取了初始化对象的参数 scaling,并没有把初始化对象传递到网格 Mesh 类下的构造函数,这种行为依赖于建模应用软件输出 AS3 Away3D 模型类的这一特定的方法,因此它并不是一个通用的方法。

```
model=new Ogre(
{
    scaling:0.01
});
model.material=modelMaterial;
model.z=100;
scene.addChild(model);
}
```

6.5 静态模型

可以使用 3DS、AWD、KMZ、ASE 和 OBJ 这些模型格式载入和直接显示静态 3D 对象。下面的一些范例显示了如何装载嵌入的或外部的以上这些模型格式文件。

6.5.1 3DS 载入嵌入文件

3DS 模型文件格式已经存在十多年了,广泛地被编写 3D 应用程序的软件和类似的 3D 引擎所支持,当很多的模型格式要求提供一个通用标准时,3DS 格式被认为是事实上的标准。

```
package
{
    import away3d.containers.ObjectContainer3D
    import away3d.core.base.Mesh;
        import away3d.core.utils.Cast;
        import away3d.loaders.Max3DS;
        import away3d.materials.BitmapMaterial;
    public class Max3DSEmbeddedDemo extends Away3DTemplate
    {
        [Embed(source="monster.3ds", mimeType="application/octet-stream")]
        protected var MonsterModel:Class;
        [Embed(source="monster.jpg")]
        protected var MonsterTexture:Class;
        public function Max3DSEmbeddedDemo ()
        {
            super();
        }
        protected override function initScene():void
        {
            super.initScene();
            var modelMaterial:BitmapMaterial=
                new BitmapMaterial(Cast.bitmap(MonsterTexture));
            var monsterMesh:ObjectContainer3D=
                Max3DS.parse(Cast.bytearray(MonsterModel),
            {
```

当嵌入一 3D 模型文件的时候,企图用 Max3DS 类从外部 images 文件载入材质是没有用的。的确,如果企图用 Max3DS 类从外部不存在的 image 文件载入材质,这时会在 3D 对象的位置显示一个错误,在下面的例子中可看见这个错误。

为了避免 Max3DS 类试图从外部文件载入 image 文件,将 autoLoadTexture 初始化对象参数设置为 false:

```
        autoLoadTexture: false,
        z: 200
    }
);
```

解析函数 parse() 将返回一个 ObjectContainer3D 对象,由 ObjectContainer3D 对象保

存的子网格,代表从 3DS 文件里载入的 3D 对象。对每个子网格使用循环,从嵌入纹理里应用到创建的材质中。

```
for each (var child:Mesh in monsterMesh.children)
    child.material=modelMasterial;
    scene.addChild(monsterMesh);
}
```

如果 autoLoadTexture 初始化对象参数没有设置为 false,将看到一个错误,如图 6-5 所示。

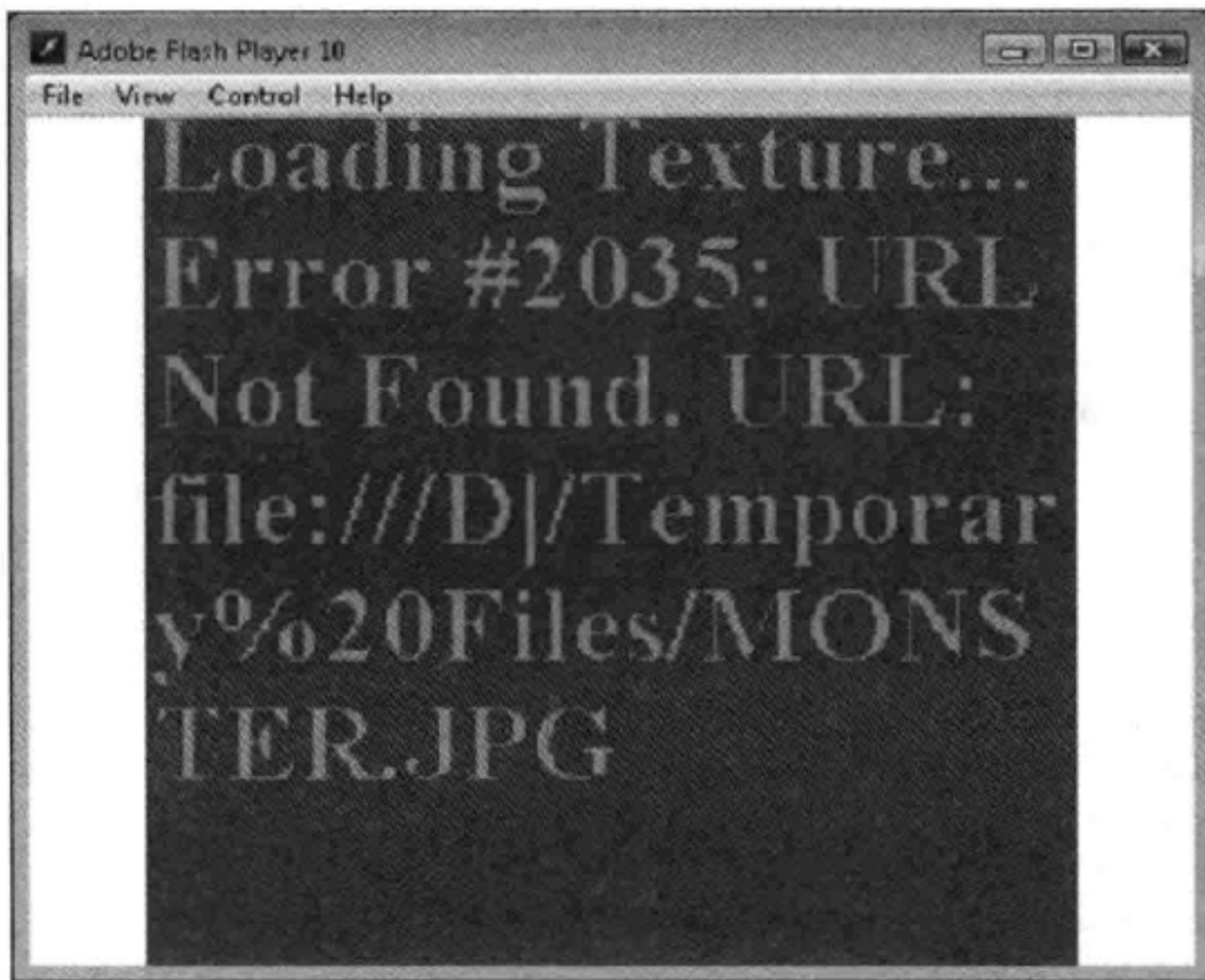


图 6-5 Adobe Flash Player 运行窗口

6.5.2 3DS 载入外部文件

从外部的 3DS 文件载入模型是非常容易的,可直接对 Max3DS load() 函数提供 3DS 文件的位置,它将通过 3DS 文件载入模型和材质。

```
package
{
    import away3d.core.utils.Cast;
    import away3d.loaders.Loader3D;
    import away3d.loaders.Max3DS;
```



```

public class Max3DSEExternalDemo extends Away3DTemplate
{
    public function Max3DSEExternalDemo ()
    {
        super();
    }
    protected override function initScene():void
    {
        super.initScene();
    }
}

```

再次提醒,网址前要加斜杠。

```

var monsterMesh:Loader3D=Max3DS.load("./monster.3ds",
{
    z:200
});
scene.addChild(monsterMesh);
}
}
}

```

6.5.3 AWD 载入嵌入文件

AWD 的模型文件格式是特别为 Away3D 使用而设计的,它是一个基于 ASCII 码的格式,这就意味着它可用正规的文本编辑器查看。

```

package
{
    import away3d.core.base.Object3D;
    import away3d.core.utils.Cast;
    import away3d.loaders.AWData;
    import away3d.materials.BitmapMaterial;
    import away3d.core.base.Mesh;
    import away3d.containers.ObjectContainer3D
    public class AWDEmbeddedDemo extends Away3DTemplate
    {
        [Embed(source="monster.awd", mimeType="application/octet-stream")]
        protected var MonsterModel:Class;
        [Embed(source="monster.jpg")]
        protected var MonsterTexture:Class;
        public function AWDEmbeddedDemo ()
        {
            super();
        }
    }
}

```

```
protected override function initScene():void
{
    super.initScene();
    var modelMaterial:BitmapMaterial=
        new BitmapMaterial(Cast.bitmap(MonsterTexture));
```

解析函数 `parse()` 将返回一个 `Object3D` 对象,在这种情况下,`Object3D` 对象实际上是一个 `ObjectContainer3D` 类的实例,于是使用 `as` 语句,把返回的 `Object3D` 对象投放到 `ObjectContainer3D` 对象。

```
var monsterMesh:ObjectContainer3D=
    AWData.parse(Cast.bytearray(MonsterModel),
    {
        z: 200
    }
    ) as ObjectContainer3D;
```

使用 `for` 循环语句,检查每个 `ObjectContainer3D` 的子网格 `Mesh`:

```
for each (var object:Object3D in monsterMesh.children)
```

再次使用 `as` 语句,但这一次把 `ObjectContainer3D` 对象的子网格投放到 `Mesh` 类。

```
var mesh:Mesh=object as Mesh;
```

如果上面的赋值语句成功(即 `mesh` 变量不为 `null`),那么使用嵌入资源建立的材质赋值给 `mesh`:

```
if(mesh != null)
    mesh.material=modelMaterial;
}
scene.addChild(monsterMesh);
}
}
```

`AWData` 类将试图从外部的 `images` 用 `BitmapFileMaterial` 类载入 `AWD` 文件里引用的材质,这里没有停止这种行为的选项,这就意味着,如果 `SWF` 文件没有正确的访问权限,会看见抛出的异常信息。然而这不是一个大问题,因为只有 `Adobe Player` 的 `debug` 版本才能显示这种异常信息,对绝大多数终端用户不会看见这个警告的信息。

可能会看见的异常信息如下:

```
SecurityError:Error# 2148:SWF file file:///D:/Temporary%20Files/
AWDEmbeddedDemo.swf cannot access local resource file:///D\
Temporary%20Files/monster.jpg. Only local-with-file system and trusted local SWF files may access
local resources.
```

```
at flash.display::Loader/get content()
```

```
at away3d.materials::BitmapFileMaterial/onComplete ( ) [c:\Away3D\away3d\materials\
BitmapFileMaterial.as:62]
```

由于 AWD 文件的格式是基于 ASCII 码的,因此可以用文本编辑器打开 AWD,通过删除对外部 image 文件的引用,来解决这个问题。

图 6-6 是原 AWD 文件,monster.jpg 文字突出地显示在第 7 行。

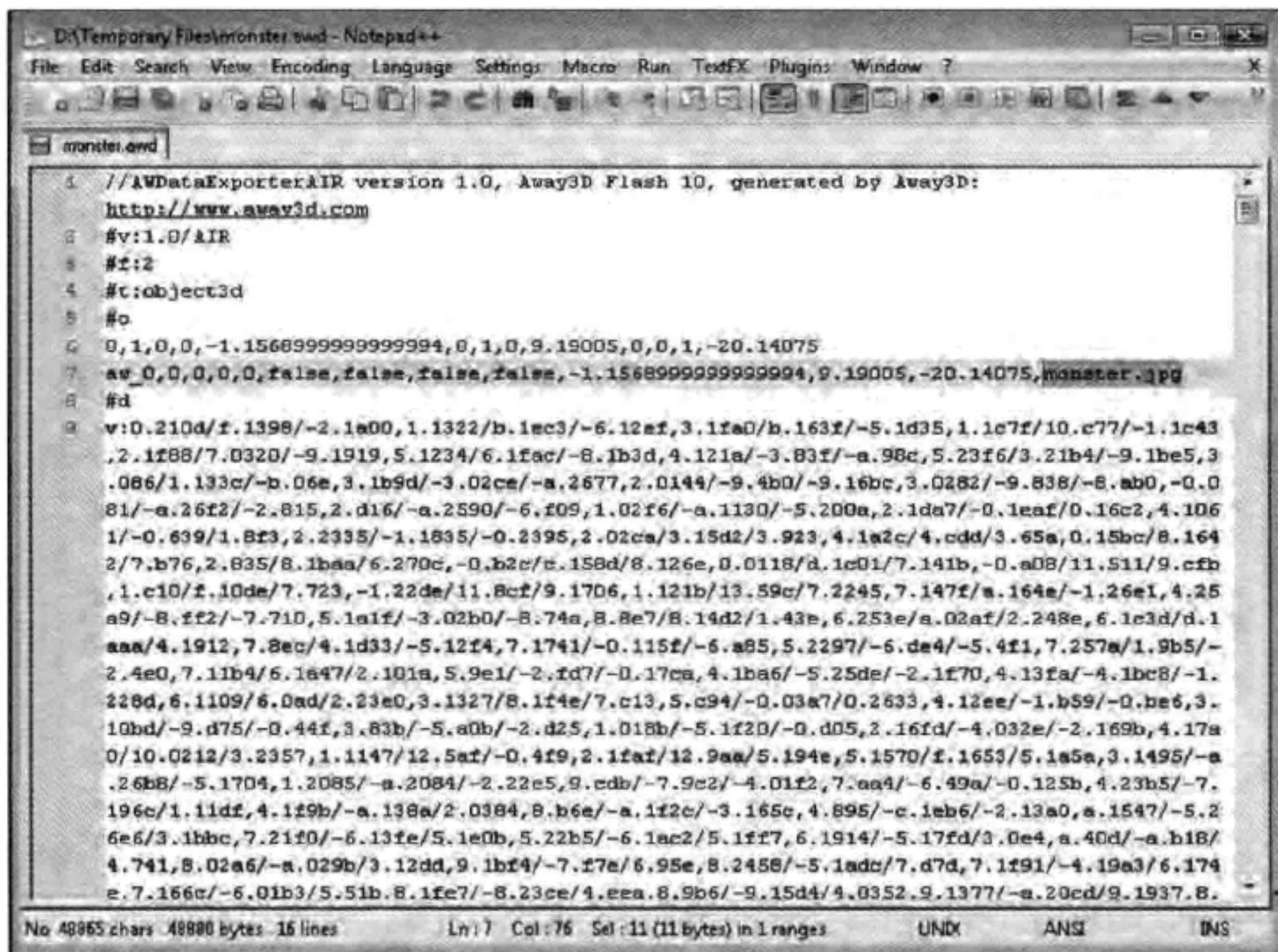


图 6-6 AWD 文件的格式

后面紧跟的 monster.jpg 文字被删除了。

如果使用删除 JPG 文件引用后的新 AWD 文件嵌入并载入,AWData 类就不会试图载入外部的 image 文件,这就会停止抛出异常。

6.5.4 AWD 载入外部文件

当载入外部 AWD 文件时,不用担心像上面载入嵌入 AWD 文件时的材质问题,直接用 AWData 类去载入并使用 AWD 文件里引用的纹理即可。

```
package
```

```

{
import away3d.loaders.AWData;
import away3d.loaders.Loader3D;
public class AWDEExternalDemo extends Away3DTemplate
{
    public function AWDEExternalDemo ()
    {
        super();
    }
    protected override function initScene():void
    {
        super.initScene();
    }
}

```

一定要在网址前加斜杠。

```

var monsterMesh:Loader3D=Max3DS.load("./monster.awd",
{
    z:200
});
scene.addChild(monsterMesh);
}
}

```

6.5.5 ASE 载入嵌入文件

ASE 模型文件格式,是 3ds Max 使用的,它使用 ASCII 字符(不像 3DS 的二进制格式),这就表明,用文本编辑器打开它是可读的。

使用嵌入的 ASE 模型文件,是相当简单的。事实上,以其他的模型文件格式载入一个嵌入模型文件时,必须意识到默认的情况下文件所引用的纹理如何从外部文件载入。但是使用 Ase 类,就没有这样麻烦的工作,或用特别的初始化对象参数来处理纹理问题。

```

package
{
    import away3d.core.base.Mesh;
    import away3d.core.utils.Cast;
    import away3d.loaders.Ase;
    import away3d.materials.BitmapMaterial;
    public class ASEEmbeddedDemo extends Away3DTemplate
    {
        [Embed(source="monster.ase", mimeType="application/octet-stream")]
        protected var MonsterModel:Class;
    }
}

```



```

[Embed(source="monster.jpg")]
protected var MonsterTexture:Class;
public function ASEEmbeddedDemo()
{
    super();
}
protected override function initScene():void
{
    super.initScene();
    var modelMaterial:BitmapMaterial=
    new BitmapMaterial(Cast.bitmap(MonsterTexture));
    var monsterMesh:Mesh=
        Ase.parse(Cast.bytearray(MonsterModel),
        {
            z:50
        }
    );
    monsterMesh.material=modelMaterial;
    scene.addChild(monsterMesh);
}
}
}

```

6.5.6 ASE 载入外部文件

载入一个外部 ASE 文件,分为以下两步来完成,第一步就是用 Ase 的载入函数 load() 以通常的方法载入文件:

```

package
{
    import away3d.core.base.Mesh;
    import away3d.events.Loader3DEvent;
    import away3d.loaders.Ase;
    import away3d.loaders.Loader3D;
    import away3d.materials.BitmapFileMaterial;
    public class ASEExternalDemo extends Away3DTemplate
    {
        public function ASEExternalDemo()
        {
            super();
        }
        protected override function initScene():void
        {

```

```
super.initScene();
```

一定要在网址前加斜杠。

```
var monsterMesh:Loader3D=Ase.load("./monster.ase",
{
    z:200
});
```

第二步就是手工载入材质,Ase 类并不解析 ASE 文件格式里的材质信息,为了适应这一情况,就要在上面的文件载入完成事件时,启动另一个载入纹理的过程。

当处理 Loader3DEvent. LOAD_SUCCESS 事件的响应函数被触发时,会调用 Loader3D 的 addOnSuccess()函数,通过它注册一个 onLoadSuccess()函数,由它启动下载 BitmapMaterial 材质。

```
monsterMesh.addOnSuccess(onLoadSuccess);
scene.addChild(monsterMesh);
}
```

在 onLoadSuccess()函数中,使用 BitmapFileMaterial 载入一个外部纹理,并应用到 3D 对象。

```
protected function onLoadSuccess(event:Loader3DEvent):void
{
    (event.loader.handle as Mesh).material=
        New BitmapFileMaterial("monster.jpg");
}
}
```

6.5.7 OBJ 载入嵌入文件

OBJ 模型文件格式,是 Wavefront 技术公司为其高级可视化的动画包开发的,后来合并到 Maya 3D 建模应用软件,它是基于 ASCII 码的,这就意味着可用一般的文本编辑器读它,OBJ 文件通常与定义材质的 3dmax 材质库文件一起使用。

```
package
{
    import away3d.core.base.Face;
    import away3d.core.base.Mesh;
    import away3d.core.utils.Cast;
    import away3d.loaders.Obj;
    import away3d.loaders.Loader3D;
```

```

import away3d.materials.BitmapMaterial;
public class OBJEmbeddedDemo extends Away3DTemplate
{
    [Embed(source="monster.obj", mimeType="application/octet-stream")]
    protected var MonsterModel:Class;
    [Embed(source="monster.jpg")]
    protected var MonsterTexture:Class;
    public function OBJEmbeddedDemo()
    {
        super();
    }
    protected override function initScene():void
    {
        super.initScene();
        var modelMaterial:BitmapMaterial=
            new BitmapMaterial(Cast.bitmap(MonsterTexture));
        var monsterMesh:Mesh=
            Obj.parse(Cast.bytearray(MonsterModel),
{
    z: 200,

```

当使用嵌入的 OBJ 模型文件格式时,设置 useMtl 初始化对象参数为 false 是非常重要的,如果 useMtl 初始参数为 true(这是默认值),则将试图下载通常伴随 OBJ 模型文件的 MTL 文件。而试图下载一个不存在的 MTL 文件,将导致显示一个错误情景,或抛出一个异常。

```

useMtl: false
}
} as Mesh;

```

由于从嵌入的 OBJ 文件构造 3D 对象方法的原因,必须对每个 3D 对象的 Face 设置使用的材质,而不是把材质赋给 Mesh material 属性。

```

    for each (var face:Face in monsterMesh.faces)
        face.material=modelMaterial;
    scene.addChild(monsterMesh);
}
}
}

```

如果 useMtl 初始化对象参数没有设置为 false,可能会看到如图 6-7 所示的错误例子。

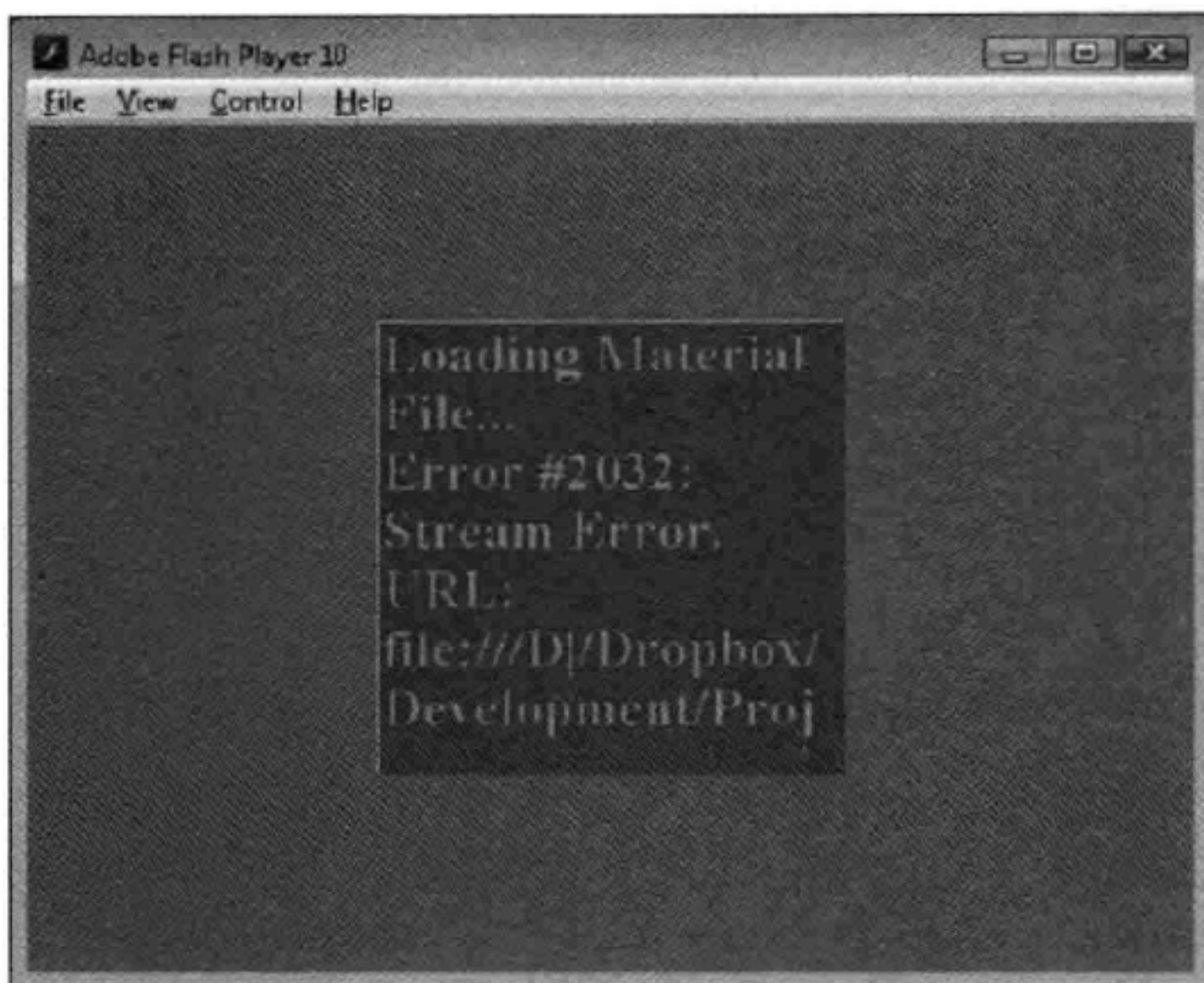


图 6-7 Adobe Flash Player 错误提示窗口

6.5.8 OBJ 载入外部文件

当载入一个外部的 OBJ 模型格式文件时,Obj 类试图载入用 3dmax 材质库定义的与 OBJ 类同名的材质,而其他方面载入一个外部的 OBJ 文件是相当简单的。

如果 MTL 文件的属性里,凡是以空格键或 Tab 键开始的一行,OBJ 类就不会对它解释,有些模型输出文件里,如 3ds Max 会把空格键或 Tab 键加到 MTL 文件的属性里,可以用文本编辑工具打开它,将空格键或 Tab 键删除。

```
package
{
    import away3d.loaders.Loader3D;
    import away3d.loaders.Obj;
    public class OBJExternalDemo extends Away3DTemplate
    {
        public function OBJExternalDemo()
        {
            super();
        }
    }
}
```



```
protected override function initScene():void
{
    super.initScene();
}
```

确保在网址前加斜杠。

```
var monsterMesh:Loader3D=Obj.load("./monster.obj",
{
    z:200
});
scene.addChild(monsterMesh);
}
```

6.5.9 初始化对象的使用问题

贯穿本书的很多地方,都使用了一个术语“init object”,经常用文字标记符号建立一个初始化对象。虽然它们相应于 Init 类,但 init object 类与 Init 类是不同的。

Init 类的实例保存对 init object 类的引用,并且提供了很多方法,能很容易地读出 init object 的属性。如此,一个 init object(注意首字母是小写“i”),包含很多用来设置对象的初始化值。而 Init 对象(注意首字母是大写“I”)是 Init 类的一个实例,它提供了一种方便的方式读出 init object 的属性。

到目前为止,并没有直接使用 Init 类,但是弄清它是如何工作的,让我们知道从嵌入文件里载入 3D 对象使用材质,与第 2 章里原始 3D 对象使用的材质,两者之间的不同是很重要的。也许读者已经注意到了,当使用模型载人类时(如 Md2 类),是直接通过 Mesh material 属性使用材质,而第 2 章里,用于原始 3D 对象的材质是使用初始化对象的 material 参数。

为了阐述这些不同,首先必须知道 Init 类是如何工作的,下面以 Init 类提供的 getMaterial() 函数来看一看。

```
public function getMaterial(name:String):Material
{
    if (init==null)
        return null;
    if( ! init.hasOwnProperty(name))
        return null;
    var result:Material=Cast.material(init[name]);
    delete init[name];
    return result;
}
```

用于 `getMaterial()` 函数的逻辑,与 `Init` 类提供的其他 `get` 函数是相同的,它首先检查请求的 `init` 属性在 `init objec` 里存不存在,若不存在,则返回 `null`,若存在,则将这个属性(`init[name]`)放到 `Material` 变量里,在返回 `material` 变量前,刚访问过的初始化对象 `init objec` 的属性(`init[name]`)被删除了。

因为读一次 `init object`,它的初始化属性就被删除了,再调用 `Init` 类的 `get` 函数读取 `init` 属性,实际上请求了一个被删除掉了的属性。这就表明,如果两个对象共享同一个 `init object`,第一个对象读取指定的初始化对象属性(通过 `Init` 类),是唯一一次能获得分配给属性的值。当看到同一 `init object` 要通过很多类传递的时候,初始化对象的属性在第一次读取时被删除了,这一点很重要。下面的序列列出了初始化对象提供给网格 `Mesh` 的解析函数 `parse()` 将要经历各个类名,按先子类,子类再调用父类,最后将取得的参数依次逐级反向返回的顺序列出。

- `Md2.parse`
- `Loader3D.parse`
- `loaderCube constructor`
- `loader3D constructor`
- `ObjectContainer3D constructor`
- `Mesh constructor`
- `Object3D constructor`
- `AbstractParse constructor...`

正如所看到的这样,初始化对象提供给网格 `Mesh` 的解析函数 `parse()` 多次传递,每次,初始化对象的属性都有消失的机会,更准确地说,就是发生在初始化对象的 `material` 属性上。上面的第 6 步,初始化对象的 `material` 属性消失了,这一步,`material` 属性用于定义原始组件立方体 `Cube` 的材质,立方体 `Cube` 用在当载入 `MD2` 文件并解析它的存放地点时。然后在第 9 步,第二次请求 `material` 属性,把它用于定义载入 3D 对象的材质,当然,这时 `material` 属性消失了,`material` 属性不再有效。

这里所述的问题发生在很多 `Away3D` 类请求同一个初始化对象的相同属性的时候,并且属性消失的命令不是立即就很明显的,在 `MD2` 类的情况里,`material` 初始化对象的属性消失是发生在网格 `Mesh` 的构造函数 `constructor` 用于存放原始立方体 `Cube` 地方的时候,而不是在 `AbstractParse constructor` 里,它是想把材质用于载入的 3D 对象的(这就是当提供一个初始化对象的 `material` 属性的时候,可能看到的这种情况)。

解决这一问题的方法,就是直接对指定的属性赋值,这就是为什么在这一章的很多例子里,把请求的材质直接赋给网格 `Mesh` 的 `material` 属性的原因。

6.6 把载入的模型转换成 ActionScript 类

网格 Mesh 对象有一个函数叫 `asAS3Class()`, 它能把 3D 对象转储成 ActionScript 类。

下面用原先在 MD2EmbeddedDemo 例子里的 `initScene()` 函数, 建立一个新的 MD2 文件里的网格 Mesh 对象, 然后用跟踪函数 `trace()` 输出 `asAS3Class()` 函数返回的字符串。

```
protected override function initScene():void
{
    super.initScene();
    md2Mesh=Md2.parse(Cast.bytearray(MD2Model));
    trace(md2Mesh.asAS3Class("Ogre","",true,true));
}
```

`asAS3Class()` 函数对复杂的模型, 能够输出几兆字节的数据, 可以找到一个 Vizzy Flash Tracer(<http://code.google.com/p/flash-tracer>) 工具, 当处理如此大的跟踪转储的时候, 比自己编写的工具要方便得多。

把 3D 模型转换到 ActionScript 类, 提供了某种程度上的防复制的保护作用, 它使利用第三方软件复制解压提取 3D 模型是困难的, 并且因为导出的 ActionScript 类, 仅包含 Away3D 所请求的数据, 这是因为通过删除一些在 3D 文件格式里没有关联的数据能导出比较小的模型文件(虽然不是总是这种情况。对于更详细的细节, 请看第 13 章中介绍的将 3D 对象保存为 ActionScript 类的好处)。

当使用 Away3D 3.6 版的 `asAS3Class()` 函数的时候, 动画不能输出, 这一功能的实现估计会包含在 Away3D 的新版中。

作为一种选择, 可以使用图如 6-8 所示的 preFab 工具, 输入 3D 模型文件, 然后输出为一个 AS 类, preFab 是一个免费的工具, 它运行在 Adobe AIR 的平台上, 并能从 <http://www.closier.nl/prefab/> 下载。

使用 preFab 把 3D 模型文件变换成 ActionScript 类的步骤如下。

- (1) 单击 File|Import 3D 命令导出模型;
- (2) 选择希望变换的 3D 模型文件, 然后单击 OK 按钮;
- (3) 这时, 可以看到 Geometry integrity report 窗口, 单击 Close 按钮, 返回到主窗口;
- (4) 在输入的 3D 模型上单击, 这时 3D 模型被蓝色框包围;



图 6-8 preFab 运行窗口

- (5) 单击 Export|Export to Aawy3D AS3 class 命令；
- (6) 选择 Selected Object 和 Only Geometry 选项；
- (7) 在 ClassName 文本框中输入类的名字；
- (8) 可以不指定 Package 或指定这个类属于的 Package；
- (9) 单击 Save File 按钮；
- (10) 选定文件保存的位置,再单击 Save 按钮。

照 相 机

在第1章里简略地接触过 Camera3D 类,就像实际生活中的照相机, Camera3D 类有很多能修改的属性,例如, focus、zoom 和 field。在这一章里,将看到这些属性如何影响照相,以及它们能用于与各种 Away3D 里的 lens 类连接。

Away3D 里还包含很多其他的照相机类,它们提供一种很容易的方法跟踪一个运动着的 3D 对象,它总是保持着一个特定时刻看到的 3D 对象,或是从不同的角度所看到的 3D 对象。

本章主要内容:

- 照相机类的属性
- 照相机可使用的不同镜头类
- 可用的不同的照相机类

7.1 照相机类的属性

在摄影时,调整照相机镜头焦距的长度,就修改了它的视角(Field Of View, FOV),也称为视域,从图 7-1 中,能看见照相机工作的情况。

正如所看到的,短焦距增加了照相机的视域,而长焦距减小了照相机的视域,较大的视域,照相机捕获的场景范围大,因为照片的物理尺寸不变,这必然使场景中的每个对象在照片中显得小些。当视域较小的时候,也是同样道理,照相机捕获的场景范围小些,在场景中的每个对象在照片中就显得大些。

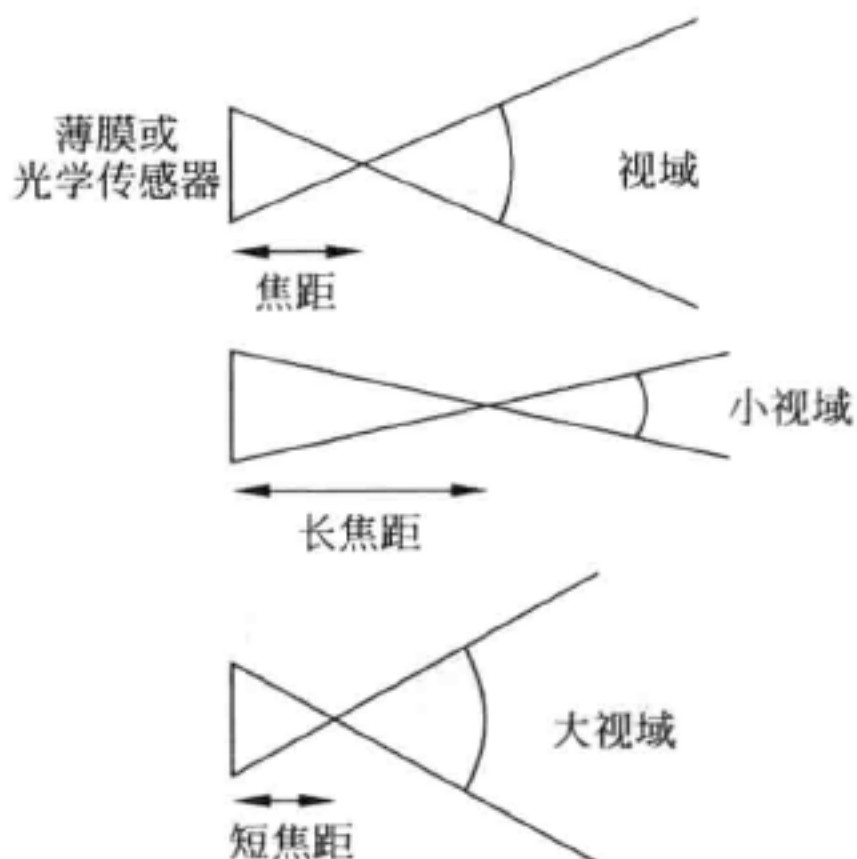


图 7-1 照相机焦距长度与视域的关系

Camera3D 类有一个 focus 属性,它与焦点长度有相似的属性,当 focus 的属性值增大时,这个视域(用 fov 属性表示)减小,缩小了场景可视的范围,并且扩大了可视 3D 对象在场景上的大小。相反地,如果减小 focus 的属性值,则增大了视域,将使场景可视范围大些,这样,减小了可视 3D 对象在屏幕上的大小。

Camera3D 类还包含一个 zoom 属性,增大 zoom 属性值,将拉进照相机的焦距,放大是由减小照相机的视域完成的,这就是 zoom 属性在 Camera3D 类里的工作原理。同理,减小 zoom 属性值,将会增大照相机的视域。

fov 属性本身能直接被修改,并且如此做出的修改,也将相应地修改 zoom 属性值,也就是增大 fov 属性值,将减小 zoom 属性值,反之,减小 fov 属性值,将增大 zoom 属性值。

当使用透视镜头 PerspectiveLens、正交镜头 OrthogonalLens 和球面镜头 SphericalLens 的时候, zoom 和 focus 这两个属性可交替地使用。ZoomFocusLens 类,依据 zoom 或 focus 给定的值,画出不同的场景,虽然一般来讲差异是很小的。

7.2 照相机的镜头

就像实际照相机一样, Away3D 的照相机类,能通过各种不同的镜头观察场景,在 Away3D 里有 4 种镜头类是可用的,每个都包含在 away3d.cameras.Lenses 包里。

(1) 放大焦距镜头 ZoomFocusLens。

- (2) 透视镜头 PerspectiveLens。
- (3) 正交镜头 OrthogonalLens。
- (4) 球面镜头 SphericalLens。

把上面的镜头类用于照相机对象,就是把镜头类分配给照相机的 lens 属性,代码如下:

```
camera.lens=new SphericalLens();
```

7.2.1 放大焦距镜头和透视镜头类

ZoomFocusLens 类是指定给照相机镜头属性的默认类,这个类画的场景非常像人们所感觉到的真实世界,虽然在大多数情形下,ZoomFocusLens 类能恰当地绘画场景,但这个类是从较早的编码保留下来的。现在编写的 3D 应用程序中,PerspectiveLens 类是更通常使用的绘画场景的一种方法。这两个类只有一些微小的区别,但是正如在第 8 章里将要介绍到的,在有些场合下,使用 PerspectiveLens 类是必需的。

图 7-2 展示的是用 ZoomFocusLens 类或 PerspectiveLens 类观察到的场景表现情形。

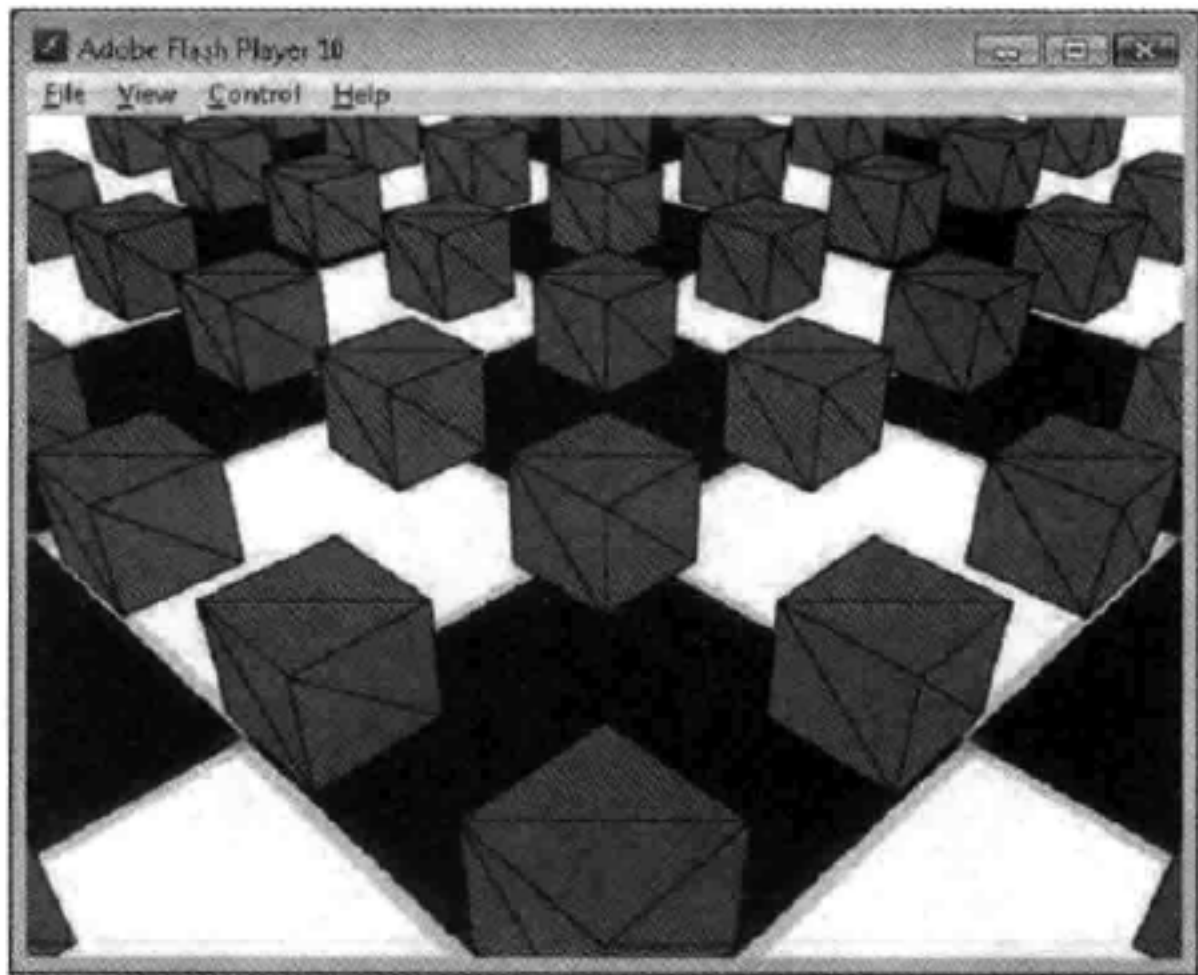


图 7-2 放大焦距镜头或透视镜头观察的场景

7.2.2 球面镜头类

当视域增大时,用 ZoomFocusLens 类或 PerspectiveLens 类的设计方法,会导致一个更扭曲的视图,通过图 7-3 将会看到这一点,它是在大的视域里,通过 PerspectiveLens 类所看到的场景,注意到显示在屏幕底部的立方体全都被画歪斜了。

对于这种情况,使用 SphericalLens 类很合适,这个球面镜类用于替换广角镜头,有时



图 7-3 大视域中透视镜头观察的场景

候,把它叫做鱼眼镜头类,它看到的场景,犹如从球面反射镜表面反射出的景象,并且它避免了扭曲,如图 7-4 所示。

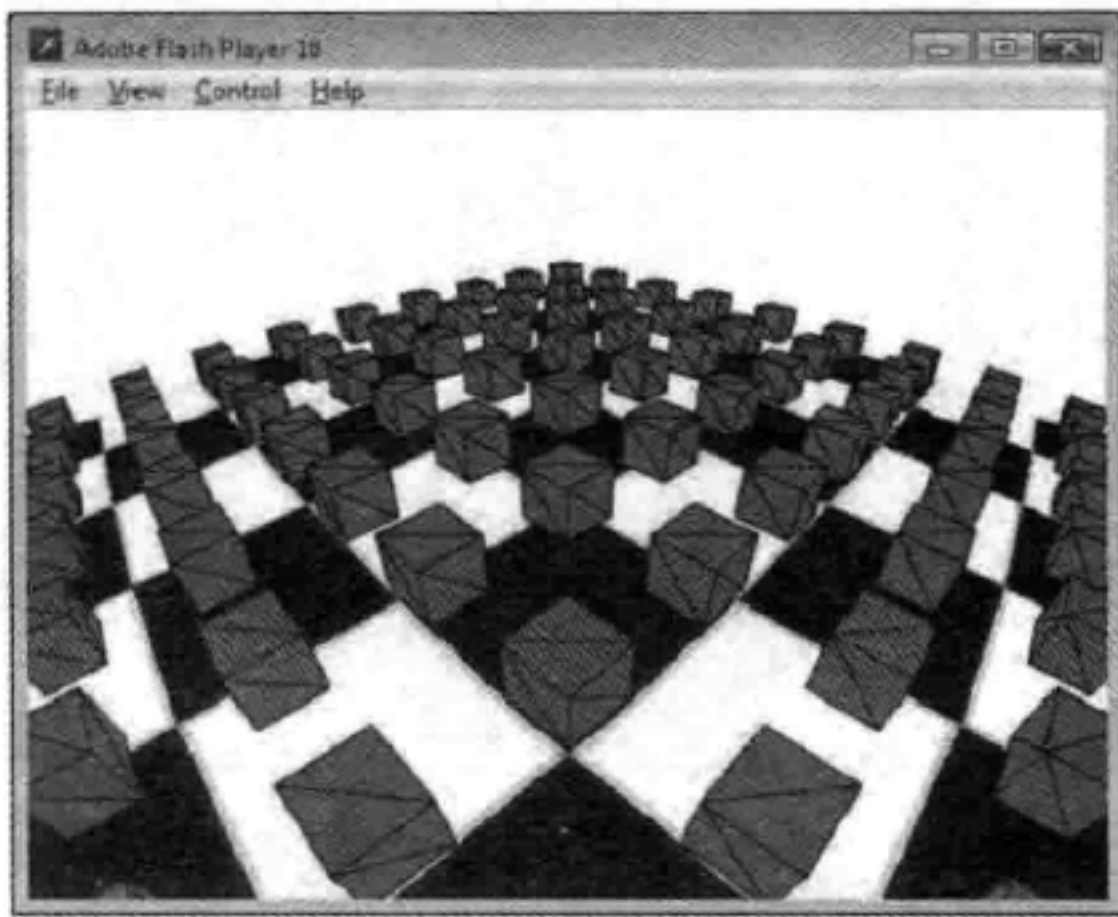


图 7-4 球面镜头观察的场景

7.2.3 正交镜头类

镜头类中的 ZoomFocusLens 类、PerspectiveLens 类和 SphericalLens 类里的有些方法,在构建一个 3D 透视图时,是用假设当 3D 对象到照相机的距离增大时,按比例缩小 3D 对象

建立的,这一效果与人类眼睛对世界的感觉类似。但是在某些情况下,这一效果是人们所不希望的,等轴投影能用于替换上述情况,它绘画出的 3D 对象用相同大小,彼此间维持平行线等距的方法。等轴投影用于实时方案和高级游戏,以及计算机辅助设计(CAD)。

在 Away3D 中,正交镜头类用于画出等体积的视图,正如图 7-5 所示,当画到屏幕上时,尽管它们到照相机的距离不同,立方体的大小仍保持一致,且它们的两边彼此间维持平行。

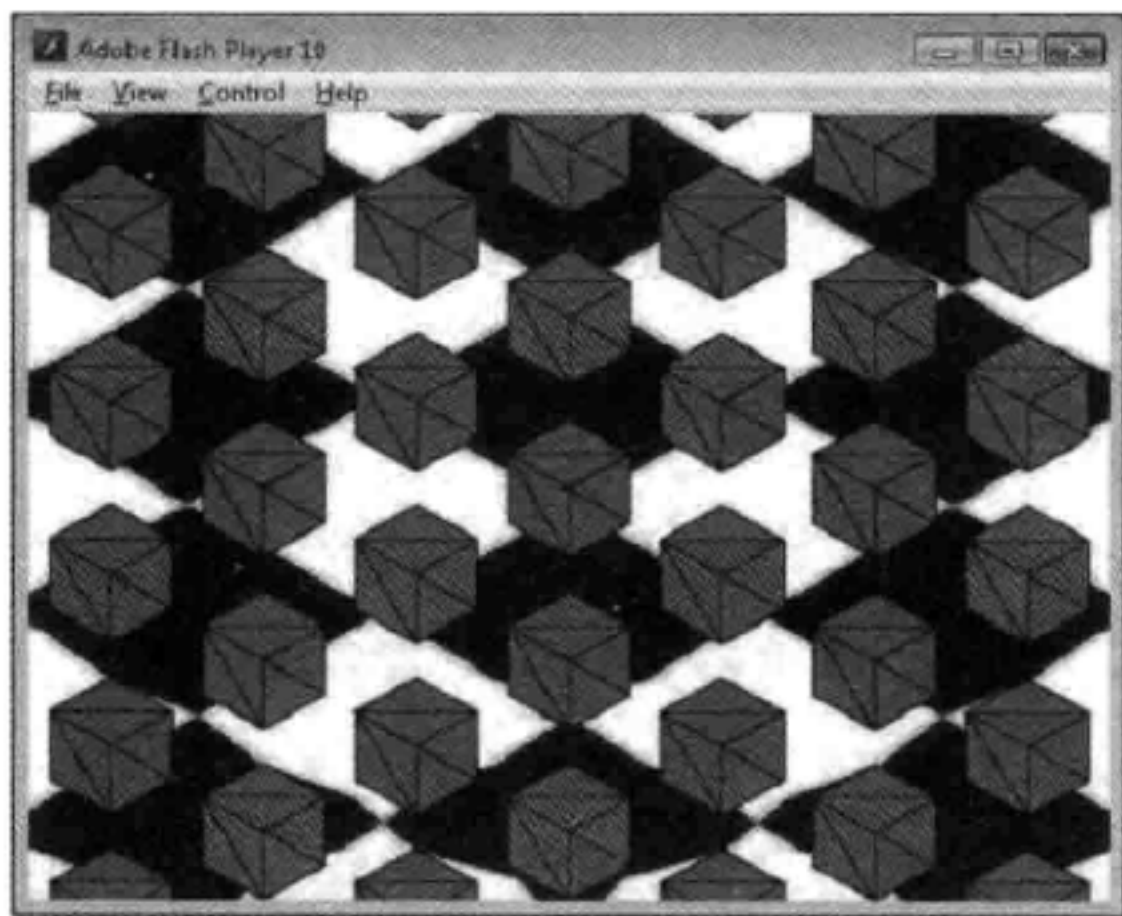


图 7-5 正交镜头观察的场景

7.3 照相机类

Away3D 包含很多照相机类,在第 1 章所介绍的 Away3DTemplate 类里,已经使用过 Camera3D 类。Camera3D 类在场景中能定位、转动,但是它不能够很容易实现对于一个 3D 对象进行跟随、滑动或跟踪。取代这些功能的是 TargetCamera3D、HoverCamera3D 和 SpringCam 类。

为了示范这三种照相机类的使用方法,建立一个叫做 CameraDemo 的应用程序,这个应用程序可使我们使用这些照相机类,用来观察在场景里运动的原始球体,并能响应键盘的输入。

```
package  
{
```

照相机类,可在 Away3d.cameras 包里找到:

```
import away3d.cameras.HoverCamera3D;  
import away3d.cameras.SpringCam;
```

```
import away3d.cameras.TargetCamera3D;
import away3d.core.clip.FrustumClipping;
import away3d.core.render.Renderer;
import away3d.core.utils.Cast;
import away3d.materials.BitmapMaterial;
import away3d.primitives.Plane;
import away3d.primitives.Sphere;
import flash.events.Event;
import flash.events.KeyboardEvent;
import flash.events.MouseEvent;
public class CameraDemo extends Away3DTemplate
{
```

把嵌入的纹理文件,应用到地面(由原始 plane 表示),给运动球提供一个参照点,

```
[Embed(source="checkerboard.jpg")]
protected var CheckerBoardTexture:Class;
```

sphere 属性引用了添加到场景里的 Sphere 的原型。

```
protected var sphere:Sphere;
```

在这个应用程序中,使用了三种相机类(旋转相机 HoverCamera3D、目标相机 TargetCamera3D 和跟踪相机 SpringCam),每个都用与它们名字相同的属性来引用。

```
protected var hoverCamera:HoverCamera3D;
protected var springCamera:SpringCam;
protected var targetCamera:TargetCamera3D;
```

lastStageX 和 lastStageY 属性用于保存最近帧的鼠标的位置,这样就能计算出给定的帧中,鼠标从最近帧到给定帧移动了多远。

```
protected var lastStageX:Number;
protected var lastStageY:Number;
```

当鼠标按钮按下时,mouseButtonDown 属性设置为 true,反之设置为 false。

```
protected var mouseButtonDown:Boolean;
```

展示 SpringCam 和 TargetCamera3D 的最好方法是跟踪一个运动的目标。当按下键盘上相应的箭头键时,与运动目标相对应的 4 个运动属性设置为 true,释放箭头键时为 false。这样可通过在每帧里用很小的输入量,响应键盘的输入使球运动和旋转。

```
protected var moveForward:Boolean;
protected var moveBackward:Boolean;
protected var turnLeft:Boolean;
protected var turnRight:Boolean;
```

```
public function CameraDemo()  
{  
    super();  
}  
protected override function initEngine():void  
{  
    super.initEngine();  
}
```

在这个演示里,默认的渲染器能引起球和地面间的景深分类出现问题,使用四叉树渲染器提供的方法可简单地解决这问题。

```
view.renderer=Renderer.CORRECT_Z_ORDER;
```

也可能遭遇一些麻烦,尽管地板在屏幕上可见,但有些部分可能在视图里被裁掉了,把新的 FrustumClipping 对象,赋值给视图的 Clipping 属性,确保仅仅地板的那些不可见的部分被剪掉。

```
view.clipping= new FrustumClipping():  
}
```

initScene()函数用于建立一个球体原始模型和平面原始模型,平面原始模型表示地板,并把这两个对象添加到场景中,棋盘的材质用于平面原始模型,它提供给我们当球围绕场景运动时的参考点。

```
protected override function initScene():void  
{  
    super.initScene();  
    sphere=new Sphere(  
    {  
        radius: 10,  
        y: 10  
    }  
);  
scene.addChild(sphere);  
var plane:Plane=new Plane(  
    {  
        material:new BitmapMaterial(Cast.bitmap(CheckerBoardTexture)),  
        width: 500,  
        height: 500  
    }  
);  
scene.addChild(plane);  
}
```

然后调用 addHoverCamera()函数,这样把旋转照相机配置为初始化的照相机。

```
addHoverCamera();  
}
```

侦听函数 `initListeners()` 代码中包含针对很多不同事件的处理函数。对于这个例子,我们必须侦听,按下键盘上的键事件 (`KeyboardEvent.KEY_Down`) 和键释放时的事件 (`KeyboardEvent.KEY_UP`), 鼠标按键按下时的事件 (`MouseEvent.MOUSE_DOWN`) 和鼠标按键释放时的事件 (`MouseEvent.MOUSE_UP`), 以及当鼠标移动时的事件 (`MouseEvent.MOUSE_MOVE`)。

```
protected override function initListeners():void
{
    super.initListeners();
    stage.addEventListener(
        MouseEvent.MOUSE_DOWN,
        onMpuseDown
    );
    stage.addEventListener(
        MouseEvent.MOUSE_UP,
        onMpuseUp
    );
    stage.addEventListener(
        MouseEvent.MOUSE_MOVE,
        onMpuseMove
    );
    stage.addEventListener(
        KeyboardEvent.KEY_DOWN,
        onKeyDown
    );
    stage.addEventListener(
        KeyboardEvent.KEY_UP,
        onKeyUp
    );
}
protected override function onEnterFrame(event:Event):void
{
    super.onEnterFrame(event);
```

如果旋转照相机是当前的照相机,必须调用它的 `hover()` 函数,更新旋转照相机的位置和朝向,使旋转照相机朝向指定给它的目标角度,以便用来响应鼠标的移动。

```
if(hoverCamera != null) hoverCamera.hover();
```

同样地,如果跟踪照相机是当前的照相机,必须访问它的 `view` 属性,这个功能与旋转照相机的 `hover()` 函数相同,并移动照相机响应跟随 3D 对象的运动。

```
if(springCamera != null) springCamera.view;
```

依靠按下键盘上的对应箭头键,球将使用 `moveForward()` 和 `moveBackward()` 函数运

动,并用 yaw()函数旋转。

```
if(moveForward) sphere.moveForward(5);
else if(moveBackward) sphere.moveBackward(5);
    if(turnLeft) sphere.yaw(-5);
else if(turnRight) sphere.yaw(5);
}
```

当按下鼠标按键的时候,MouseEvent.MOUSE_DOWN事件将被启动,并调用onMouseDown()函数。

```
protected function onMouseDown(event:MouseEvent):void
{
```

设置 mouseButtonDown 属性为 true,表明当前的鼠标按键被按下。

```
mouseButtonDown=true;
```

使用 lastStageX 和 lastStageY 两个属性保存当前鼠标的位置,这样可计算出鼠标移动到下一帧有多远。

```
lastStageX=event.stageX;
lastStageY=event.stageY;
}
```

当释放鼠标按键的时候,MouseEvent.MOUSE_UP事件将被启动,并调用onMouseUp()函数。

```
protected function onMouseUp(event:MouseEvent):void
{
```

mouseButtonDown 属性为 false,表明当前的鼠标按键已释放。

```
mouseButtonDown=false;
```

当按下键盘上的箭头键时,KeyboardEvent.KEY_DOWN事件被激发,它调用onKeyDown()函数来处理这一事件,在这里设置 moveForward、moveBackward、turnLeft 和 turnRight 的属性为 true,假设与它们相对应的键被按下。

```
protected function onKeyDown(event:KeyboardEvent):void
{
    switch (event.KeyCode)
    {
        case 38:                //Up Arrow
            moveForward=true;
            break;
        case 40:                //Down Arrow
```

```

        moveBackward=true;
        break;
    case 37:                //Left Arrow
        turnLeft=true;
        break;
    case 39:                //Right Arrow
        turnRight=true;
        break;
    }
}

```

当释放键盘上的箭头键时,KeyboardEvent.KEY_UP 事件被激发,它调用 onKeyUp() 函数来处理这一事件,在这里设置 moveForward、moveBackward、turnLeft 和 turnRight 的属性为 false,假设与它们相对应的键被释放。

```

protected function onKeyUp(event:KeyboardEvent):void
{
    switch (event.KeyCode)
    {
        case 38:            //Up Arrow
            moveForward=false;
            break;
        case 40:            //Down Arrow
            moveBackward= false;
            break;
        case 37:            //Left Arrow
            turnLeft= false;
            break;
        case 39:            //Right Arrow
            turnRight= false;
            break;
    }
}

```

我们也侦听键盘上的数字键 1、2 和 3 键的释放,当这些键中的一个键被释放的时候,就调用 addHoverCamera()、addSpringCamera()或 addTargetCamera()函数,来初始化它所对应的观看场景的新照相机类型。

```

    case 49:                //1
        addHoverCamera();
        break;
    case 50:                //2
        addSpringCamera();
        break;
    case 51:                //3
        addTargetCamera();
        break;
}

```

```
}  
}
```

余下的这些代码,涉及建立、更新这三种不同照相机的内容。

7.3.1 目标照相机

目标照相机的表现,除了它始终在其视野中心对准某一特定的 3D 对象之外,就像一个人们日常使用的照相机一样。当键盘上的数字键 3 释放的时候就引发调用 addTargetCamera()函数建立并激活目标照相机 TargetCamera。

```
protected function addTargetCamera():void  
{
```

当把新的照相机添加到场景的时候,对其他的两个照相机的引用要设置为空 null。

```
    hoverCamare=null;  
    springCamera=null;
```

目标照相机由 TargetCamera3D 类代表。下面的代码将建立一个新的 TargetCamera3D 类的实例,对它的初始化函数提供初始的对象,以便给出相机对准的 3D 对象,以及给定相机的全局位置(0,100,0)把相机放置到 Y 轴正方向的 100 单位处。

提供初始的对象 Target,是唯一能被 TargetCamera3D 类识别的参数,然而像所有的照相机类一样,TargetCamera3D 类也是继承自由相机 Camera3D 类的,因此对自由相机 Camera3D 类提供的初始参数也传递到了 TargetCamera3D 类的构造函数。

```
    targetCamera=new TargetCamera3D(  
    {  
        target: sphere,  
        y: 100  
    }  
);
```

为了通过新相机观察场景,把它赋给 View3D 的相机属性。

```
    View3D.camera= targetCamera;  
}
```

表 7-1 列出了能为 TargetCamera3D 类所识别的初始化对象的参数。

表 7-1 TargetCamera3D 类构造函数的参数

参 数	数 据 类 型	默 认 值	说 明
target	Object 3D	new Object3D()	定义对准的 3D 对象

7.3.2 旋转照相机

与目标照相机一样,旋转照相机始终对准 3D 对象,但是它能围绕 3D 对象移动,仿佛旋转照相机是围绕 3D 对象的一个椭球体表面滑动一样,这就提供了一个可从任何角度观察 3D 对象的方法,而且通过响应鼠标移动它能建立一个接口而操控观察的场景。

```
protected function addHoverCamera():void
{
    targetCamare=null;
    springCamera=null;
```

旋转照相机由 HoverCamera3D 类代表,这里建立一个新的该类实例,并给它提供初始化的对象,以便使相机总能看见目标 3D 对象和相机放置位置与 3D 目标间的距离,以及相机能用的最小倾角和初始的目标倾角。

```
hoverCamera=new HoverCamera3D(
{
    target: sphere,
    distance: 100,
    mintiltangle: 5,
    tiltAngle: 45
});
view.camera=hoverCamera;
```

在这个应用程序里,旋转相机的位置是用鼠标控制的。我们已经弄清了鼠标 mouseButtonDown 的属性值是因响应鼠标按键的按下或释放通过 onMouseDown() 和 onMouseUp() 两个函数设置为 true 或是 false 的。现在,当响应 MouseEvent.MOUSE_MOVE 事件的时候,就调用 onMouseMove() 函数处理鼠标移动事件,通过定义这个处理鼠标移动事件的函数来修改旋转相机的位置,

```
protected function onMouseMove(event:MouseEvent):void
{
```

如果 mouseButtonDown 的属性值是 true,这就表明鼠标按键已经按下了,而且 hoverCamera 属性不是 null 表明旋转相机是当前的照相机,这时就用鼠标的移动来修改旋转相机。

```
If(mouseButtonDown && hoverCamera != null)
{
```

旋转相机摆动的角度(围绕 Y 轴转动),是由鼠标在屏上横跨移动的水平距离计算的,

即由当前的鼠标水平位置(event.stageX),减去上一帧的鼠标水平位置(lastStageX),得出从上一帧以来鼠标水平移动了多远。

```
var pan:int=(event.stageX-lastStageX);
```

旋转相机上下倾斜的角度(围绕 X 轴转动),可用同样的方法得到,只不过这次用鼠标当前的和上次的鼠标位置的垂直距离,计算出鼠标从上一帧以来垂直地移动了多远。

```
var pan:int=(event.stageY-lastStageY);
```

旋转相机有两个属性,定义它围绕 3D 目标对象应有的位置,第一个是 panAngle,这个属性定义了旋转相机围绕 Y 轴要求的角度;第二个属性 tiltAngle,定义了相机围绕 X 轴要求的角度。给这两个属性赋值,并不能使相机立即转到一个新位置,而是在每次调用 hover() 函数时,逐渐更新相机的位置和朝向。于是,在 onEnterFrame() 函数里,每帧都调用 hover() 函数一次,旋转相机立即到达所要求的用 panAngle 属性和 tiltAngle 属性定义的新位置。

```
hoverCamera.panAngle += pan;
hoverCamera.tiltAngle += tilt;
```

然后,将鼠标的当前位置保存到 lastStageX 和 lastStageY 属性中。

```
lastStageX=event.stageX;
lastStageY=event.stageY;
}
}
```

表 7-2 中列出了 HoverCamera3D 类能识别的初始化对象参数。

表 7-2 HoverCamera3D 类构造函数的参数

参数	数据类型	默认值	说 明
yfactor	Number	2	定义相机水平朝向与垂直朝向时,距离不同的分数,较高的值,表示当相机在目标的上面或下面时,离目标更远些
distance	Number	800	相机到 3D 目标的距离,当 tiltangle 是零时,yfactor 属性能用来改变相机的垂直移动
wrapPanAngle	Boolean	false	定义当 pan 角大于 360°或小于 0°时,是否回绕
panAngle	Number	0	相机绕 Y 轴转动的角度
tiltAngle	Number	90	相机绕 X 轴转动的角度
minTiltAngle	Number	-90	tiltAngle 的最小界限
maxTiltAngle	Number	90	tiltAngle 的最大界限
steps	int	8	每次调用 hover() 函数时的分级步数,大的值将使相机每次移动小的距离

7.3.3 跟踪照相机

跟踪照相机由 SpringCam 类代表,用于跟随 3D 对象在场景里的运动,好像 3D 对象附上了这个相机一样。

```
protected function addSpringCamera():void
{
    targetCamare=null;
    hoverCamera=null;
    springCamera=new SpringCam();
    springCamera.target=sphere;
    view.camera= springCamera;
}
}
```

SpringCam 类与其他的照相机类不一样,它的初始属性不能通过一个初始对象来指定,接受初始对化对象的构造函数,将传递到它的基类 Camera3D 的构造函数,但是给 SpringCam 类指定的全部属性必须在建立一个新的 SpringCam 类时逐个设定,表 7-3 列出了 SpringCam 类公开的公共属性。

表 7-3 SpringCam 类构造函数的参数

参 数	数 据 类 型	默 认 值	说 明
target	Object3D	null	定义相机跟随的 3D 目标对象,如果为 null,其行为同自由相机 camera3D
stiffness	Number	1	弹性的刚度,定义伸缩性,较高值表示跟踪目标对象,更多地修改到目标的距离
damping	Number	4	定义弹性的内磨擦,它影响弹性的快速返回,较高值将更多地减少相机的弹跳
mass	Number	40	相机的质量,较高值将增加相机对 3D 对象的拖曳阻力,给相机移动更大的动量
positionOffset	Vector3D	Vector3D(0,5,-50)	重新设置相机相对于 3D 目标对象的位置
lookOffset	Vector3D	Vector3D(0,2,10)	设置相机相对于 3D 目标对象看见的位置

鼠标互动性

几乎每一个 Flash 应用程序,都用鼠标作为接受用户输入的主要手段。在 Flash 中通过对指定的鼠标事件注册一个函数,来响应鼠标事件。Away3D 也遵循同样的原理,并且确实,3D 对象上执行的鼠标事件的名字与 Flash 在 2D 上执行的鼠标事件的名字相同。这就表示,凡是那些在传统的 2D Flash 应用程序中,使用过鼠标的任何开发者,在 Away3D 的应用程序中做同样的事,都不会有什么困难。

在本章中也将会看见,在屏幕上的鼠标位置能投射到一个 3D 的场景里,它提供了建立 3D 的鼠标拖曳接口,这种接口在很多游戏中都有过。

本章主要内容:

- Away3D 支持的鼠标事件
- ROLL_OVER/ROLL_OUT 和 MOUSE_OVER/MOUSE_OUT 之间的事件的区别
- 把鼠标位置投射到场景

8.1 Away3D 鼠标事件

相对于 3D 对象,Away3D 有很多支持它的鼠标事件,所有这些事件在 MouseEvent3D 类里定义为一个字符串常数。这些常数列在表 8-1 中。

表 8-1 MouseEvent3D 类的鼠标事件字符串及其功能

MouseEvent3D 鼠标事件字符串	说 明
MOUSE_MOVE	当鼠标光标划过一个 3D 对象表面时,触发此事件
MOUSE_OVER	当鼠标光标移动到一个新 3D 对象上或新材质的表面上时,触发此事件
MOUSE_OUT	当鼠标离开 3D 对象或表面时,触发此事件
MOUSE_DOWN	当鼠标按键按下,而这时光标又在 3D 对象上,触发此事件
MOUSE_UP	当鼠标按键释放,而这时光标又在 3D 对象上,触发此事件
ROLL_OVER	当鼠标光标移动到还不是属于光标下的这一组内的 3D 对象上时,激活此事件
ROLL_OUT	当鼠标光标离开 3D 对象时,触发此事件

任何一个继承 Object3D 的包含场景的类,都能触发这些事件,view 除 MouseEvent. ROLL_OVER 和 MouseEvent. ROLL_OUT 外,也能触发这些事件。

如果曾经用过 Flash 的鼠标事件,那么对 Away3D 的鼠标事件也应该是熟悉的,因为 Away3D 的鼠标事件是 Flash 事件的写照,响应事件的方法也类似于传统的 Flash 鼠标事件的处理方法。

使用 addEventHandler()注册一个管理事件的函数:

```
myObject3D.addEventHandler(
MouseEvent3D.MOUSE_MOVE,
onMouseMove
);
```

从上面的代码可知管理事件的函数侦听 MouseEvent3D.MOUSE_MOVE 事件的发生,一旦事件发生,管理函数便立即调用鼠标事件处理函数 onMouseMove。

鼠标事件处理函数看起来如下:

```
Function onMouseMove(event:MouseEvent3D):void
{
    // do something here
}
```

传递给管理事件的 addEventHandler()函数的 MouseEvent3D 类(来自于 away3d.events 包)包含一些唯一的属性,允许在 3D 里用鼠标事件来进行编程工作,这些属性列在表 8-2 中。

表 8-2 鼠标事件的属性及其功能

属 性	说 明
screenX	在视口坐标里,发生事件的水平坐标
screenY	在视口坐标里,发生事件的垂直坐标
screenZ	在视口坐标里,发生事件的景深坐标

续表

属 性	说 明
sceneX	在全局场景坐标里,发生事件的 x 坐标
sceneY	在全局场景坐标里,发生事件的 y 坐标
sceneZ	在全局场景坐标里,发生事件的 z 坐标
view	发生事件的视口对象
object	发生事件的 3D 对象
elementVO	发生事件的 3D 元素对象
material	发生事件的 3D 元素的材质对象
uv	发生事件的 3D 元素的 UV 坐标
ctrlKey	指定 Ctrl 键是否激活,true 或 false
shiftKey	指定 Shift 键是否激活,true 或 false

8.2 ROLL_OVER/ ROLL_OUT 和 MOUSE_OVER/ MOUSE_OUT 之间的区别

Away3D 支持 MouseEvent3D. ROLL _ OVER/MouseEvent3D. ROLL _ OUT, MouseEvent3D. MOUSE_OVER 和 MouseEvent3D. MOUSE_OUT 这两对事件,当鼠标移动到一个 3D 对象上面而后离开时,触发这些事件。当调度处理事件函数时,两对事件间稍为有些不同。

用下面的应用程序建立两个重叠的球体,把它们添加到一个容器里面,并用 trace() 函数跟踪 MouseEvent3D. ROLL _ OVER、MouseEvent3D. ROLL _ OUT、MouseEvent3D. MOUSE_OVER 和 MouseEvent3D. MOUSE_OUT 这些事件,当处理一个事件函数被容器调度时,通知我们。

```
package
{
    import away3d.containers.ObjectContainer3D;
    import away3d.events.MouseEvent3D;
    import away3d.materials.WireColorMaterial;
    import away3d.materials.WireframeMaterial;
    import away3d.primitives.Sphere;
    public class MouseRollMoveEventDemo extends Away3DTemplate
    {
        public function MouseRollMoveEventDemo()
        {
            super();
        }
    }
}
```

在初始化场景函数 `initScene()` 中,建立了两个球体,沿 X 轴方向 100 单位,把它们分开,由于它们都有默认的 100 单位的半径,相互将可覆盖。

```
protected override function initScene():void
{
    super.initScene();
    var sphere1:Sphere=new Sphere(
    {
        x: 50,
        y: 0,
        z: 500
    }
    );
    var sphere2:Sphere=new Sphere(
    {
        x: -50,
        y: 0,
        z: 500
    }
    );
}
```

把两球 sphere 3D 对象添加到 `ObjectContainer3D` 容器里,作为它的子对象,然后,把容器作为场景子对象加到场景里。

```
var container:ObjectContainer3D=new ObjectContainer3D(sphere1, sphere2);
scene.addChild(container);
```

为 `MouseEvent3D.MOUSE_OVER`、`MouseEvent3D.MOUSE_OUT`、`MouseEvent3D.ROLL_OVER` 和 `MouseEvent3D.ROLL_OUT` 这 4 个事件建立侦听器,在响应这些事件时,调用一个匿名函数 `trace()` 来显示一行文本。

```
container.addEventListener(
    MouseEvent3D.MOUSE_OVER,
    function(event: MouseEvent3D):void
    {
        trace("Container Mouse Over");
    }
);
container.addEventListener(
    MouseEvent3D.MOUSE_OUT,
    function(event: MouseEvent3D):void
    {
        trace("Container Mouse OUT");
    }
);
```

```
);  
    container.addEventListener(  
        MouseEvent3D.ROLL_OVER,  
        function(event: MouseEvent3D): void  
        {  
            trace("Container Mouse Roll Over");  
        }  
    );  
    container.addEventListener(  
        MouseEvent3D.ROLL_OUT,  
        function(event: MouseEvent3D): void  
        {  
            trace("Container Mouse Roll Out");  
        }  
    );  
}  
}
```

随着上面程序的运行,把鼠标从围绕两球的空白处移动到左边的球体上面,如图 8-1 所示。

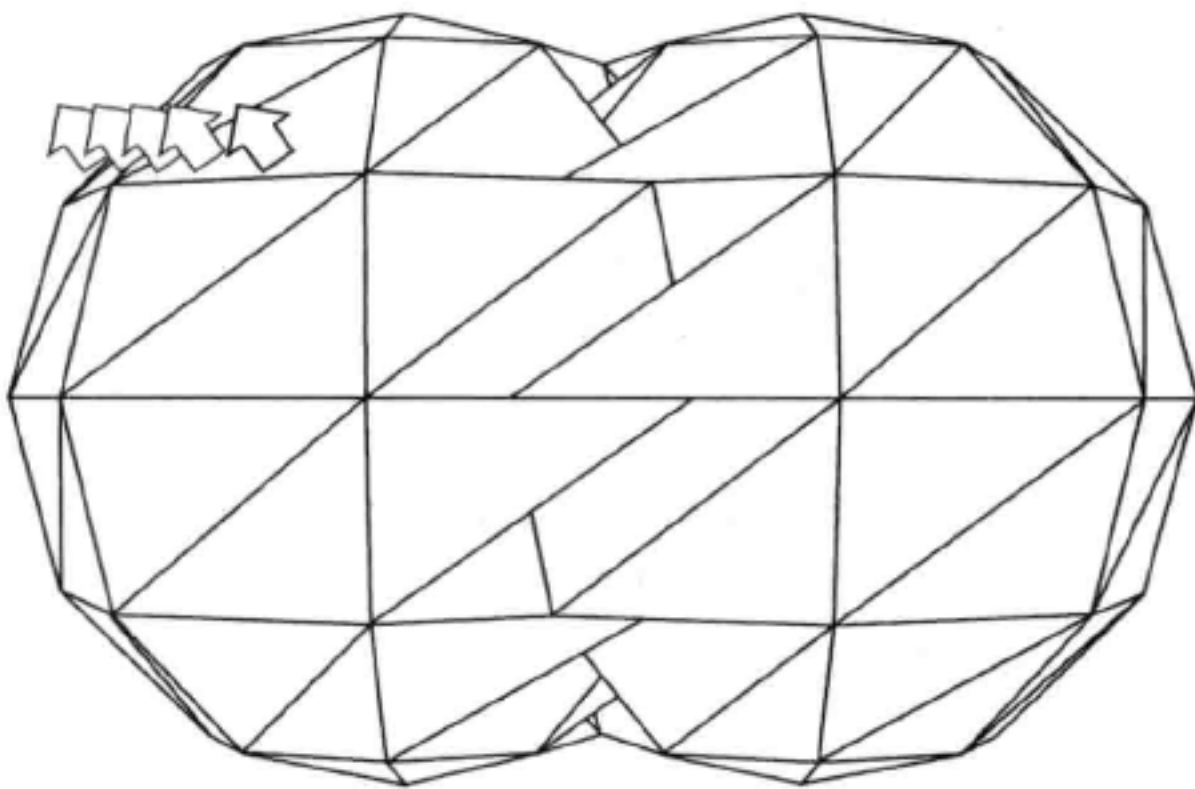


图 8-1 鼠标从空白处移到球体

这个过程产生下面的输出:

- Container Mouse Over
- Container Roll Over

这是因为这两个事件 `MouseEvent3D.ROLL_OVER` 和 `MouseEvent3D.MOUSE_OVER` 都触发了,这就是当光标移动到 3D 对象上的时候,期望能看见的输出。

现在,当鼠标从左边的球移到右边的球的时候,如图 8-2 所示。

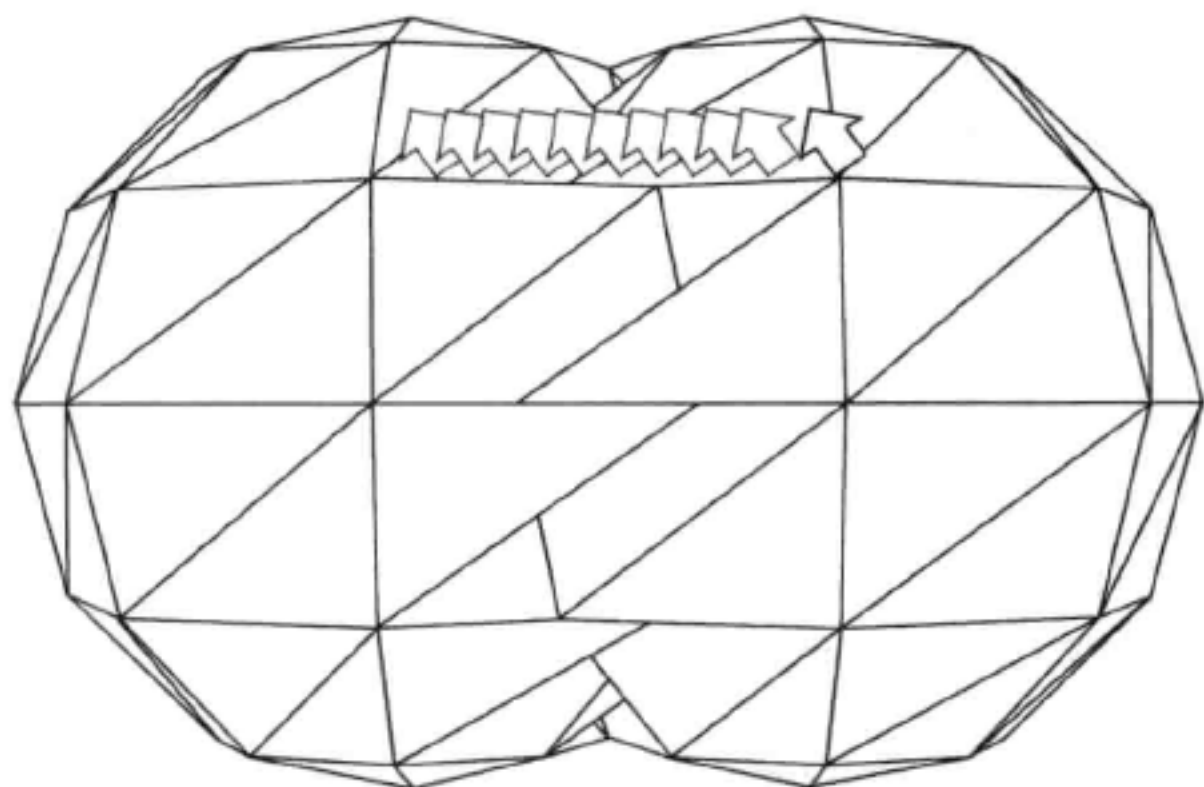


图 8-2 鼠标在球体间移动

第二个移动产生下面的输出：

- Container Mouse Out
- Container Mouse Over

当鼠标移出左边球的时候,MouseEvent3D.MOUSE_OUT 事件派出,这个事件向上传到父容器。因为这两个球有重叠的部分,当鼠标移出左边球的同时,它便立即到了右边球的上面,MouseEvent3D.MOUSE_OVER 事件再次派出,这两个事件接二连三地又向上传到父容器。

移动期间,光标从容器里的一个子 3D 对象到旁边一个子 3D 对象,但它绝不能到达两者间的空白空间的上面,换句话说,鼠标绝不能移动到容器的外面,两对事件间存在很大的不同就在于:当鼠标移到各个单独的子对象的上面的时候,MouseEvent3D.MOUSE_OVER 和 MouseEvent3D.MOUSE_OUT 事件从容器里 3D 子对象接二连三触发,仅当鼠标移动到全部子对象的上面和从全部子对象上面移出的时候,MouseEvent3D.ROLL_OVER 和 MouseEvent3D.ROLL_OUT 事件才触发。

最后,鼠标移出右边的球,返回到空白空间上,如图 8-3 所示。

最后的移动产生以下的输出：

- Container Mouse Out
- Container Roll Out

因为鼠标移动到了右边球的外面,MouseEvent3D.MOUSE_OUT 事件被触发,又因为鼠标移动到了容器内所包含的全部子对象的外面,MouseEvent3D.ROLL_OUT 事件也被触发。

这样,当鼠标移动到容器里的任何一个子对象的上面的时候,MouseEvent3D.ROLL_OVER 事件被触发,当鼠标从容器内的全部子对象上移出的时候,MouseEvent3D.ROLL_OUT 事件被触发。而在另一方面,当鼠标移动到容器内的一个子对象的上面或鼠标移出

一个子对象的时候,MouseEvent3D.MOUSE_OVER 或 MouseEvent3D.MOUSE_OUT 事件被触发。

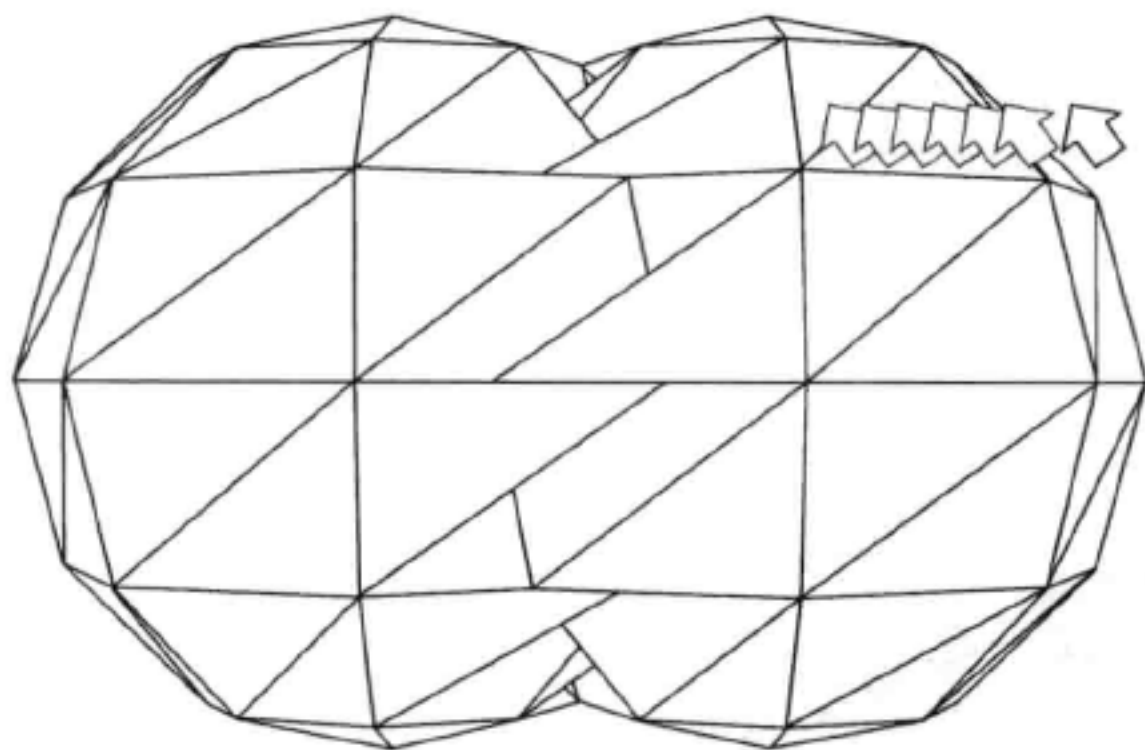


图 8-3 鼠标从球体移到空白处

8.3 将鼠标的位置投影到场景

正如所看到的,在 Away3D 中,响应来自 3D 对象的鼠标事件是非常简单的,只要指定调用定义在 MouseEvent3D 类中的各种响应事件的函数,并使用与 Flash 2D 的应用程序相同的侦听函数 addEventListener()就可以了。

除了用上述方式响应事件之外,Away3D 还可取得光标在场景里的位置,然后用这个位置构造出一条射线延伸到场景中,并与一个平面相交。图 8-4 显示了射线与一个平面相交的情况。

在下面的 InteractivityDemo 应用程序中,使用这种射线来找出平面上的一个相交点,将相交点用于改变一个球的位置,好像在场景里,球被拖曳地四处移动一样。还要响应 MouseEvent3D.MOUSE_OVER、MouseEvent3D.MOUSE_OUT 和 MouseEvent3D.MOUSE_DOWN 这些事件。

```
package
{
    import away3d.cameras.lenses.PerspectiveLens;
    import away3d.core.base.Object3D;
    import away3d.core.geom.Plane3D;
    import away3d.core.render.BasicRenderer;
    import away3d.core.utils.Cast;
    import away3d.events.MouseEvent3D;
    import away3d.materials.BitmapMaterial;
    import away3d.primitives.Plane;
```

```

import away3d.primitives.Sphere;
import flash.geom.Vector3D;
import flash.events.Event;
import flash.events.MouseEvent;
import flash.filters.GlowFilter;
public class InteractivityDemo extends Away3DTemplate
{

```

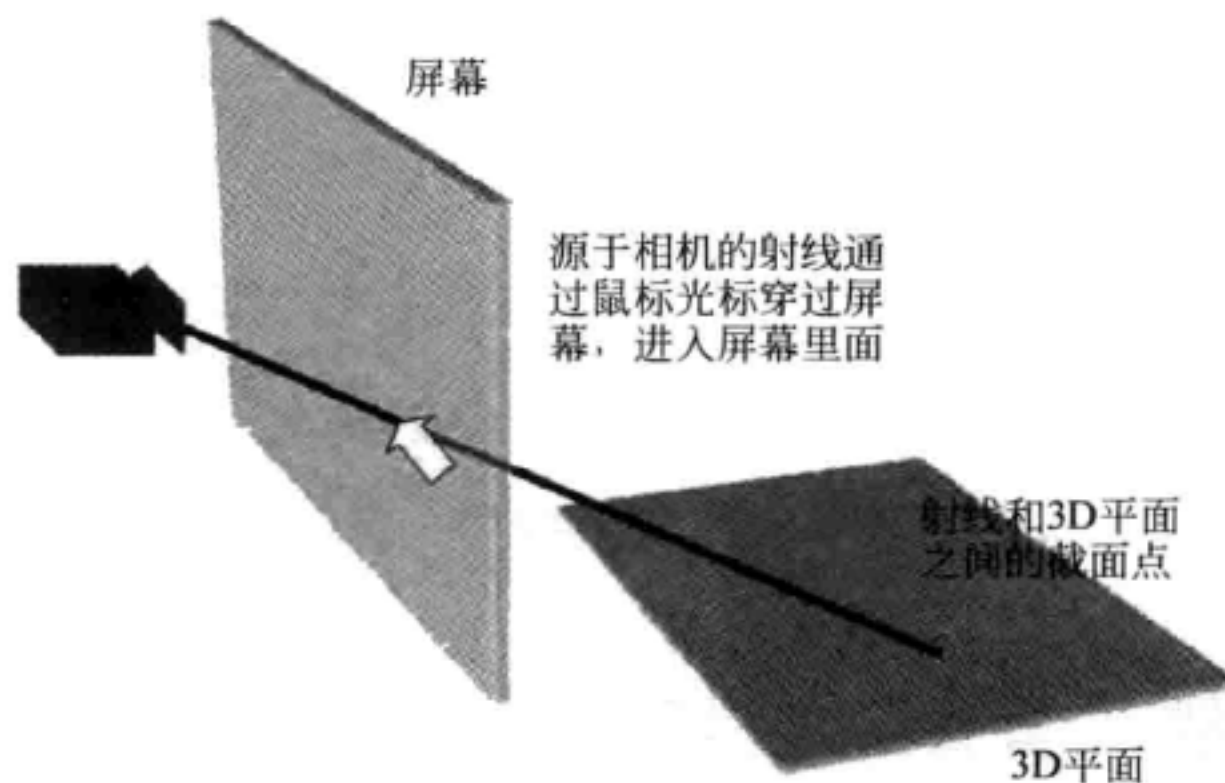


图 8-4 射线与平面相交示意图

嵌入世界象棋的棋盘 checkerboard.jpg 纹理图,作为 3D 对象的地板。

```

[Embed(source="checkerboard.jpg")]
protected var CheckerBoardTexture:Class;

```

selectedObject 属性,用于引用移动的球的 3D 对象。

```
protected var selectedObject:Object3D;
```

sphere1 和 sphere2 两个属性,用于引用添加到场景的两个球的 3D 对象。

```
protected var sphere1:Sphere;
protected var sphere2:Sphere;
```

groundPlane 属性,用于引用表示地板的平面的 3D 对象。

```
protected var groundPlane:Plane3D;
```

throughScreenVector 和 groundPosition 属性,用于引用作为射线的向量和地板 3D 平面上光标的位置。

```
protected var throughScreenVector:Vector3D;
protected var groundPosition:Vector3D;
public function InteractivityDemo()
```

```

{
    super();
}
protected override function initEngine():void
{
    super.initEngine();
}

```

用于将鼠标的位置投射到场景里,并到达平面上,只有当照相机用透视镜头时才能工作。如果使用默认的 ZooFocusLens 类的镜头,那么将要计算出光标在场景里的位置,这个位置不能与鼠标的实际位置构成一条完整的射线。

```
camera.lens=new PerspectiveLens();
```

场景包含两个球,每个都带有设置为 true 的 ownCanvas,这对于把过滤器加到各个 3D 对象上是需要的。Away3D 有个性能,叫做三角缓存(对它更详细的解释,在第 13 章中),它能导致那些带 ownCanvas 设置为 true 的 3D 对象,在它们相对于相机的距离发生变化时,不能正确地进行 z 景深排序。设置视口的属性 forceUpdate 为 true,使三角缓存不能发生作用和修改 3D 对象景深的问题。

[但是要意识到,使三角缓存不能发生作用,将对运行性能产生负面影响。]

为了看清三角缓存能使两个或多个带 ownCanvas 属性设置为 true 的 3D 对象进行景深排序的效果,把下面的这一行代码注释去掉。这时,会看到一个球总是画在另一个的前面,而不管球在场景里的什么位置。

```

view.forceUpdate=true;
}
protected override function initScene():void
{
    super.initScene();
}

```

照相机放在沿 Y 轴的正向 100 单位并稍微向下倾斜,使它能得到最好的场景视窗。

```

camera.position=new Vector3D(0,100,0);
camera.tilt(20);

```

建立 Bitmap 材质,再把它用到原始平面模型上,将平面模型添加到场景里表示地面。

```

var planeMaterial:BitmapMaterial=new
    BitmapMaterial(
        Cast.bitmap(CheckerBoardTexture)
    );
var plane:Plane=new Plane(

```

```
{
    material:planeMaterial,
    segments: 10,
    width: 1000,
    height: 1000,
    y: -15,
    z: 250
}
);
```

通过设置 screenZOffset 属性值为 1000,强迫平面画在随后将要添加的两个球的下面,第 4 章详述了 screenZOffset 属性的细节,以及很多附加的方法,这些方法能用于调整 3D 对象在场景里的景深排序的次序。

```
plane.screenZOffset=1000;
scene.addChild(plane);
```

现在,把两个球的原始模型添加到场景,ownCanvas 初始化的对象参数设置为 true,它可以使我们用色彩鲜艳过滤器突出显示在鼠标光标下的球体。第 12 章将介绍如何使用过滤器的细节。

```
sphere1=new Sphere(
{
    x: -50,
    z: 250,
    radius: 10,
    ownCanvas: true
}
);
sphere1.ownCanvas=true;
scene.addChild(sphere1);
sphere2=new Sphere(
{
    x: 50,
    z: 250,
    radius: 10,
    ownCanvas: true
}
);
scene.addChild(sphere2);
```

Plane3D 类代表一个无限平面,Plane3D 对象是不可见的对象,而且不应该与 Plane 类混淆,后者建立原始 3D 对象。

```
groundPlane=new Plane3D();
```


用法向矢量(直接指向 Y 轴正方向)和此平面上的点(在这里,是原点)来初始化 Plane3D 类,这样建立的平面位于 X 轴和 Z 轴组成的平面上。

```
groundPlane.fromNormalAndPoint(  
    new Vector3D(0, 1, 0),  
    new Vector3D()  
);  
protected override function initListeners():void  
{  
    super.initListeners();  
}
```

针对由舞台产生 MouseEvent.MOUSE_UP 的事件,注册这个事件的处理函数 onMouseUp(),注意这是一个标准的 Flash 鼠标事件处理函数,不是由任何 Away3D 类调用的。

```
stage.addEventListener(  
    MouseEvent.MOUSE_UP,  
    onMouseUp  
);
```

针对由两个球都可能发生的 MouseEvent3D.MOUSE_OVER、MouseEvent3D.MOUSE_OUT 和 MouseEvent3D.MOUSE_DOWN 的事件,注册三个对应的事件 onMouseOver()函数、onMouseOut()函数和 onMouseDOWN()函数。不像上述由舞台产生的事件,这三个事件由场景里的鼠标侦听用户动作而触发。

```
sphere1.addEventListener(  
    MouseEvent3D.MOUSE_OVER,  
    onMouseOver  
);  
sphere1.addEventListener(  
    MouseEvent3D.MOUSE_OUT,  
    onMouseOUT  
);  
sphere1.addEventListener(  
    MouseEvent3D.MOUSE_DOWN,  
    onMouseDown  
);  
sphere2.addEventListener(  
    MouseEvent3D.MOUSE_OVER,  
    onMouseOver
```

```

    );
    sphere2.addEventListener(
        MouseEvent3D.MOUSE_OUT,
        onMouseOUT
    );
    sphere2.addEventListener(
        MouseEvent3D.MOUSE_DOWN,
        onMouseDown
    );

```

`addOnMouseMove()`、`addOnMouseDown()`、`addOnMouseUp()`、`addOnMouseOver()`、`addOnMouseOut()`、`addOnRollOver()`和 `addOnRollOut()` 这些函数能用速记法,将函数连接到事件,如下面这样:

```
sphere1.addOnMouseDown(onMouseDown);
```

同样, `removeOnMouseMove()`、`removeOnMouseDown()`、`removeOnMouseUp()`、`removeOnMouseOver()`、`removeOnMouseOut()`、`removeOnRollOver()`和 `removeOnRollOut()`,能停止对事件的响应。

当鼠标移动到一个球的上面的时候,用色彩鲜艳的过滤器来突出显示它,如此就要把 `GlowFilter` 类的实例添加到一个数组,然后将它赋给 `Object3D` 类定义的 `filters` 属性。第 12 章将介绍过滤器的 `filters` 属性。

```

protected function onMouseOver(event:MouseEvent3D):void
{
    event.object.filters=[new GlowFilter()];
}

```

当鼠标移出一个球体时,则要清除过滤器,恢复到默认的样子。

```

protected function onMouseOut(event:MouseEvent3D):void
{
    event.object.filters=[];
}

```

当用鼠标在一个球体上单击的时候,这个球便赋予了 `selectedObject` 属性,即这个球被选为运动的球。

```

protected function onMouseDown(event:MouseEvent3D):void
{

```

```
selectedObject=event.object;
}
```

当鼠标键在一个球体上释放的时候, null 便赋予了 selectedObject 属性, 即两个球都没有被选中。

请注意, 上述函数取消选定球, 响应 MouseEvent.MOUSE_UP 事件, 是由 Flash 舞台发出的, 而不是由 MouseEvent3D.MOUSE_UP 事件通过 3D Object 发出的。之所以能完成这个功能, 是因为取消选定球释放鼠标按键, 是不管当释放按键时哪个 Away3D 对象是在光标下的。如果这样, 那便有这样一种可能, 当释放鼠标按键时选定的球, 根本就不在鼠标光标下, 这样如果已经注册了一个要调用的 onMouseUp() 函数, 响应 MouseEvent3D.MOUSE_UP 由原始球体发起的事件, 那就有一种可能性, 这个球不是预期要取消选定的球。

```
protected function onMouseUp(event:MouseEvent):void
{
    selectedObject=null;
}
```

在 onEnterFrame() 函数中, 把选中的球体移动到鼠标光标下的位置。

```
protected override function onEnterFrame(event:Event):void
{
    super.onEnterFrame(event);
```

如果 selectedObject 变量不为 null, 则说明两球中选择一个, 然后, 使球在鼠标光标下运动。

```
    If(selectedObject != null)
    {
```

Camera3D.unproject() 函数, 取屏幕上的二维坐标, 返回一个向量。该向量从照相机瞄准出发穿过屏幕上的二维坐标出来进到场景。

鼠标需要提供相对于舞台上视口的位置, 因为视口位于舞台的中央, 必须调整鼠标的坐标位置, 使它相对于舞台的左上角。

```
throughScreenVector=camera.unproject (
    stage.mouseX-stage.stageWidth/2,
    stage.mouseY-stage.stageHeight/2
);
```

要清楚舞台 `width/height` 和 `stageWidth/stageHeight` 这两对属性之间的区别, `width` 和 `height` 属性定义舞台的子舞台采取的范围, 而 `stageWidth` 和 `stageHeight` 属性定义了舞台本身实际的大小, 尽管它稍为有点不同, 而舞台的子舞台(像这里的 `View3D` 对象的情况)要采用舞台上的全部可用空间。但弄清这两套量度制间的不同, 也是值得的。

然后, 把照相机的位置添加到场景定位的方向矢量。

```
throughScreenVector = throughScreenVector.add(camera.position);
```

`Plane3DGetIntersectionLineNumbers()` 函数, 取两个点的位置作为参数, 照相机的位置和由照相机投射到屏幕上鼠标光标坐标的位置。返回这两点所组成的向量投射到场景中与平面相交的交点位置。

```
groundPosition =
    groundPlane.getIntersectionLineNumbers(
        camera.position,
        throughScreenVector
    );
```

然后, 将选定的球移到平面相交的交点位置。

```
selectedObject.position = groundPosition;
    }
    }
    }
}
```

当应用程序运行的时候, 将看到鼠标光标下的色彩鲜艳的球突显出来, 并可以拖曳它在场景里移动。虽然鼠标仅能在二维屏幕上移动, 但用平面上球的定位方法, 可以提供一种显示球在三维场景里移动的模拟方式。

使用精灵的特殊效果

在 Flash 程序中,精灵 Sprite 是一个绘画的对象,它通常被添加到舞台里。本书里的全部应用程序使用了第 1 章中的 Away3DTemplate 类,它就是继承了 Flash 的 sprite 类。

Away3D 也包含很多 sprite 类,尽管名字相同,但 Away3D 的 sprite 类是添加到网格 Mesh 对象里的,而不是加到 Flash 的舞台里,Away3D 的 sprite 类用于显示一个矩形的纹理,这个矩形总是面对照相机。

本章主要介绍以下三种 Away3D 中的精灵。

- (1) Sprite3D,它总是面对照相机,显示矩形上的材质。
- (2) DirectionalSprite,它根据视口里的角度,显示选定的材质。
- (3) DepthOfFieldSprite,它显示一个域的景深效果。

实际上,Sprite3D 类的使用已经在第 2 章中介绍过了,但在本章里,将看到更多的实际的这个类的实现。

因为 Sprite3D 类是相当简单的类,所以即使将很多的精灵类添加到场景中仍能维持一个合理的帧速度,这就允许大量的精灵类用于模拟粒子效果,如冒烟和爆炸。可在 <http://code.google.com/p/stardust-particle-engine> 网址里看到 Stardust 粒子系统引擎库,它可与 Away3D 集成用于模拟粒子效果。

9.1 使用 Sprite3D 类

Sprite3D 类,是所有包含在 Away3D 里的精灵类的基础,第 2 章中已经介绍过使用它的例子,在 Sprite3DDemo 类里将看到更多的实际的例子,使用 Sprite3D 类建立一个气球

上升的场景。

```
package
{
    import away3d.core.base.Mesh;
    import away3d.core.base.Vertex;
    import away3d.core.base.utils.Cast;
    import away3d.materials.BitmapMaterial;
    import away3d.sprites.Sprite3D;
    import flash.geom.Vector3D;
    import flash.events.Event;
    import flash.utils.getTimer;
    [SWF(backgroudColor="0xFFFFFFFF")];
    public class Sprite3DDemo extends Away3DTemplate
    {
```

每个气球由三个纹理中的一个表示,这里嵌入蓝色、绿色和橘红色的纹理图。

```
[Embed(source="blueballoon.png")]
protected var BlueBalloon:Class;
[Embed(source="greenballoon.png")]
protected var GreenBalloon:Class;
[Embed(source="orangeballoon.png")]
protected var OrangeBalloon:Class;
```

把每个 Sprite3D 对象加到 balloons 的集合中:

```
protected var balloons:Vector.<Sprite3D>=
    new Vector.<Sprite3D>();
```

NUMBER_OF_BALLOONS 常数,定义了有多少个气球被添加到场景里。

```
protected static const NUMBER_OF_BALLOONS:int=1000;
public function Sprite3DDemo()
{
    super();
    protected override function initScene():void
    {
        super.initScene();
```

照相机的位置设置到场景的原点:

```
this.camera.position=new Vector3D();
```

用三个 BitmapMaterial 材质对象,填充 balloonTextures 数组元素,每个是三个嵌入的纹理图中的一个。

```
var balloonTextures:Array=
```

```
[
    new BitmapMaterial(
        Cast.bitmap(BlueBalloon),
        {smooth: true}
    ),
    new BitmapMaterial(
        Cast.bitmap(GreenBalloon),
        {smooth: true}
    ),
    new BitmapMaterial(
        Cast.bitmap(OrangeBalloon),
        {smooth: true}
    )
];
```

在第2章里已经见过,在场景里 Sprite3D 对象能看见之前,必须把它加到网格 Mesh 里。这里建立一个新的网格对象 Mesh,并把它加到场景里:

```
var mesh:Mesh=new Mesh();
scene.addChild(mesh);
```

在循环语句中,建立代表这些气球的 Sprite3D 对象:

```
var sprite:Sprite3D;
for (var i:int=0; i<NUMBER_OF_BALLOONS; ++i)
{
```

建立的每个 Sprite3D 对象,随机地为其选用材质:

```
sprite=new Sprite3D(
    balloonTextures[Math.round(Math.random() *
        (balloonTextures.length -1))]
);
```

然后,把 Sprite3D 对象随机地定位在一个 $1000 \times 1000 \times 2000$ 大小的方盒子里:

```
sprite.x=Math.random() * 1000-500;
sprite.y=Math.random() * 1000-500;
sprite.z=Math.random() * 2000;
balloons.push(sprite);
```

最后,为了使 Sprite3D 对象可见,把它加到父网格对象里:

```
mesh.addSprite(sprite);
}
}
protected override function onEnterFrame(event:Event):void
{
```

```
super. onEnterFrame(event);
```

循环地通过每个 Sprite3D 对象,修改每帧的位置,以便它们表现为在一个漂浮的空间中。

```
for (var i:int=0; i<NUMBER_OF_BALLOONS; ++i)  
{
```

Sprite3D 对象的定位沿它们的 X 轴用正弦函数映射,使它们表现为轻轻地摇摆 in 微风中的样子。

```
balloons[i].x += Math.sin(getTimer()/1000+i);
```

Sprite3D 对象还要沿 Y 轴上升,当它们到达虚构的包含它们的盒子顶端的时候,它们要落到场景的底下。

```
if(balloons[i].y >= 5000)  
    balloons[i].y =- 5000;  
else  
    balloons[i].y += 5;  
}  
}  
}
```

当这个程序运行时,可看见气球海洋,如图 9-1 所示。这个简单的例子展示了 Sprite 的功能,如果想用几千个 3D Sprite 对象来代表气球,这时应用程序将很可能每秒只运行很少的几个帧,但是可以很快地、平稳地建立 Sprite3D 对象。

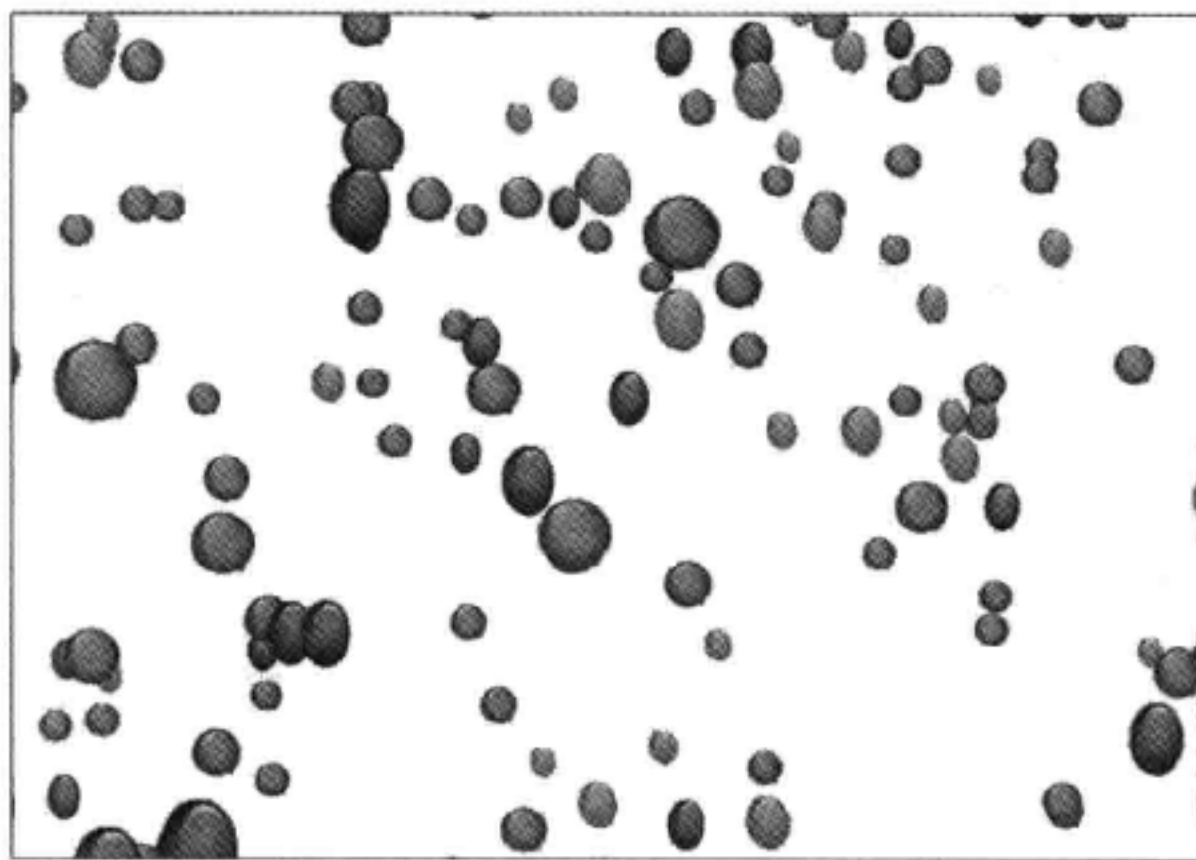


图 9-1 气球海洋程序运行效果

9.2 使用定向精灵类

定向精灵类提供了一个方法,它可以根据照相机相对 Sprite 的位置显示出所选定的材质。这样通过提供对一个复杂 3D 对象在一定视角范围内的很多快速照片,就能看见虚拟出的这个复杂 3D 对象的样子,比采用实时地绘画一个实际的 3D 对象,用了较少的处理功能。在很多老的游戏里面,如《德军司令部》、《毁灭的战士》中这种处理功能被广泛使用。

在实现 DirectionalSprite 类之前,首先必须要有很多要采用的图片或从不同的视角画出的同一个对象的很多图片。

图 9-2 为一张 3D 模型的快速照片,在这个图片里照相机位于 Z 轴的正轴方向,沿着相反的方向看这个 3D 模型,3D 模型位于坐标的原点,这样从 3D 模型的背面指向照相机的单位向量(长度为 1 的向量)是 $(0,0,1)$,这个向量是很重要的,以后要用到。

图 9-3 是一张从 3D 模型侧面看到的照片,在这个图片里,照相机位于 X 轴的正端方向,看到原点的 3D 模型,从 3D 对象到照相机的单位向量是 $(1,0,0)$ 。



图 9-2 3D 模型的 Z 轴正向效果图



图 9-3 3D 模型的侧视效果图

图 9-4 是两张 3D 对象的照片,它们是从上面的两张照片的反方向摄取的,于是从 3D 对象到照相机的单位向量分别是 $(0,0,-1)$ 和 $(-1,0,0)$ 。

现在,已经有了一些从不同的视角看到的 3D 对象的图片,下面建立一个名为 SimpleDirectionalSpriteDemo 的应用程序,用 DirectionalSprite 对象来显示它们。

```
package
{
    import away3d.core.base.Mesh;
    import away3d.core.base.Vertex;
    import away3d.utils.Cast;
```

```

import away3d.utils.Init;
import away3d.materials.BitmapMaterial;
import away3d.sprites.Directionalsprite;
import flash.geom.Vector3D;
import flash.events.Event;
public class SimpleDirectionalSpriteDemo extends Away3DTemplate
{

```

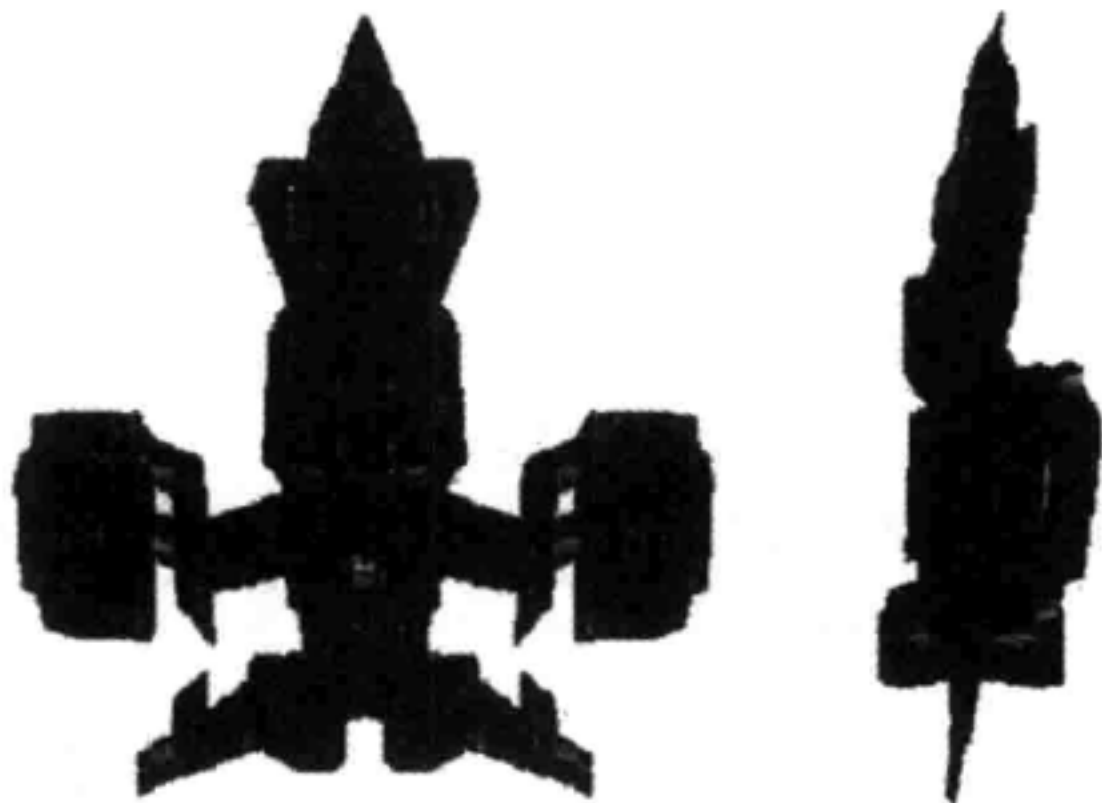


图 9-4 3D 模型的侧视和 Z 轴正向合成效果图

嵌入 3D 对象的 4 张照片：

```

[Embed(source="front.png")]
protected var FrontImage:Class;
[Embed(source="right.png")]
protected var RightImage:Class;
[Embed(source="back.png")]
protected var BackImage:Class;
[Embed(source="left.png")]
protected var LeftImage:Class;
protected var sprite:DirectionalSprite;
protected var parentMesh:Mesh;
public function SimpleDirectionalSpriteDemo()
{
    super();
}
protected override function initScene():void
{
    super.initScene();

```

这里,建立一个新的 DirectionalSprite 类的实例:

```
sprite=new DirectionalSprite();
```

DirectionalSprite 对象将显示原先展示的 4 张图像,这些图像通过 addDirectionalSprite() 函数来添加。

第一个参数,定义了从 DirectionalSprite 对象到将能观察到图像的照相机的方向,这与对准描述过的上述图像的方向相同。

第二个参数,是将要显示的材质,这里用 BitmapMaterial 对象来显示嵌入的其中一张图像。

```
sprite.addDirectionalMaterial(  
    new Vertex(0,0,1),  
    new BitmapMaterial(  
        Cast.bitmap(FrontImage),  
        { smooth: true }  
    )  
);  
sprite.addDirectionalMaterial(  
    new Vertex(1,0,0),  
    new BitmapMaterial(  
        Cast.bitmap(RightImage),  
        { smooth: true }  
    )  
);  
sprite.addDirectionalMaterial(  
    new Vertex(0,0,-1),  
    new BitmapMaterial(  
        Cast.bitmap(BackImage),  
        { smooth: true }  
    )  
);  
sprite.addDirectionalMaterial(  
    new Vertex(-1,0,0),  
    new BitmapMaterial(  
        Cast.bitmap(LeftImage),  
        { smooth: true }  
    )  
);
```

为了显示 DirectionalSprite 对象,需要把它加到 Mesh 对象里,现在来建立 Mesh 对象:

```
parentMesh=new Mesh();
```

然后,把 DirectionalSprite 对象添加到 Mesh 对象:

```
parentMesh.addSprite(sprite);
```

最后,把 Mesh 对象添加到场景:

```
scene.addChild(parentMesh);  
}
```

在 onEnterFrame() 函数里,把 DirectionalSprite 精灵围绕它自己的 Y 轴旋转,就像旋转任何其他的 3D 对象一样。注意到这一点是很重要的,以这种方式旋转 DirectionalSprite 精灵对象,实际上并没有改变精灵在场景里的方向: Away3D 的所有精灵,它们本身在全部时间里是面对照相机的,并且,这样旋转并不改变通过 DirectionalSprite 精灵显示的材质。

开始, rotationY 属性是 0,这就表示 DirectionalSprite 对象的本地坐标的 Z 轴朝向全局坐标的 Z 轴的正方向。还要记住, DirectionalSprite 对象已经位于照相机的前面,这表示,初始照相机观察到的 DirectionalSprite 对象,是从对象的后面看到的。因此,从 3D 对象指向照相机的单位向量是 (0,0,-1)。

通过 addDirectionalMaterial() 函数赋值的背面材质,我们能看见的图像,是由 BackImage 类代表的,这个类是最与照相机的相对位置匹配的。这表示,应用程序第一次运行时, DirectionalSprite 对象将显示背面图像的纹理。

增加 rotationY 属性值,有使 DirectionalSprite 对象向右转的效果,当 rotationY 属性达到 90° 的时候, DirectionalSprite 对象显示 RightImage 纹理,这是因为照相机与 DirectionalSprite 对象间的夹角和照相机与 RightImage 纹理定义的夹角相比最小。

当更多地增加照相机的 rotationY 属性值的时候, DirectionalSprite 对象将逐次显示 FrontImage 纹理、LeftImage 纹理,然后再返回显示 BackImage 纹理。

```
protected override function onEnterFrame(event:Event):void  
{  
    super.onEnterFrame(event);  
    parentMesh.rotationY +=5;  
}  
}
```

虽然这个应用程序只展示了 4 幅图像,但可使用更多的图像,完成角度间的平滑转换, DirectionalSpriteDemo 应用程序可用 72 幅图像,显示每隔 5° 的 3D 对象,这样的过程使 DirectionalSprite 3D 对象展示的旋转相当平滑,而且帧速也高。然而,当 DirectionalSprite 类用很多的图像显示 3D 对象较多细节,且维持较高的帧速时,经常使各幅图像占用的内存,比直接把 3D 对象加到场景中要多得多。

9.3 使用景深精灵类

在摄影技术中,场景景深涉及在照相机前面,当焦点对准物体时,物体前后表现清晰的区域。这个效果经常用于强调场景的一部分,而不再强调场景的前景和背景。

当画出一个在 Away3D 中场景的时候没有景深,整个场景是在完美的焦点上。然而这种效果能用 DepthOfFieldSprite 类来近似,它要预先从提供给 DepthOfFieldSprite 类的构造函数的材质里计算出很多逐渐模糊的材质图像,并将它们存储在共享的缓冲区,然后,在运行的时候,Sprite 根据 DepthOfFieldSprite 类到照相机的距离来显示它们。

```
package
{
    import away3d.containers.ObjectContainer3D;
    import away3d.core.base.Mesh;
    import away3d.core.utils.Cast;
    import away3d.core.utils.DofCache;
    import away3d.materials.BitmapMaterial;
    import away3d.sprites.DephOfFieldSprite;
    import flash.display.BitmapData;
    import flash.events.Event;
    public class DepthOfFieldSpriteDemo extends Away3DTemplate
    {
```

嵌入一个图像文件的时候,DepthOfFieldSprite 对象把它作为基础纹理。

```
[Embed(source="blackdot.png")]
protected var BlackDot:Class;
```

我们将要把 DepthOfFieldSprite 对象添加到容器 Mesh 里面,它由属性 container 来引用。

```
protected var container:Mesh;
public function DepthOfFieldSpriteDemo()
{
    super();
}
```

DofCache 类有很多属性,这些属性定义了 DepthOfFieldSprite 类的外貌,它们都定义在 initEngine() 函数里。

```
protected override function initEngine():void
{
```

```
super().initEngine();
```

属性 `aperture`, 用于近似照相机光圈的效果, `aperture` 属性的值大一些会增加域景深, 这就意味着 `DepthOfFieldSprite` 对象在照相机的前面维持一个大些的清晰区域。而较小的值, 将导致 `DepthOfFieldSprite` 对象在较小区域内显示清晰, 如图 9-5 所示。

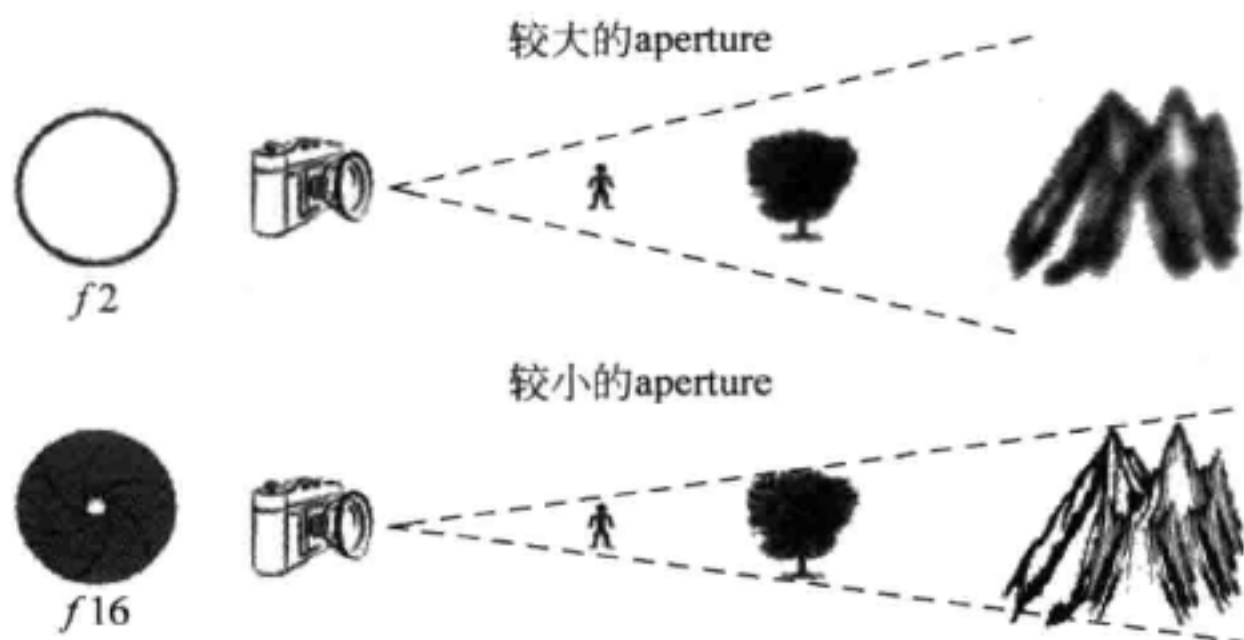


图 9-5 `aperture` 属性值大小与视域的关系

```
DofCache.aperture=50;
```

属性 `doflevels` 定义要预先计算和缓冲的 `Bitmap` 图像模糊强度的级数。这些缓冲的 `Bitmap` 图像, 在运行时, `DepthOfFieldSprite` 对象会根据它对相机的位置距离显示出来, 较大的值将使缓冲的图像从一个到下一个平滑地过渡, 虽然这提升了视觉质量, 但需要更多的内存去容纳附加的缓冲图像, 而且预先计算图像要花更多的时间。

```
DofCache.doflevels=32;
```

属性 `maxblur` 定义了 `DepthOfFieldSprite` 对象用的最大模糊强度数值, 较大的值将导致在景深域里的 `DepthOfFieldSprite` 对象与不在景深域里的 `DepthOfFieldSprite` 对象间, 有更大的判断区别。

```
DofCache.maxblur=50;
```

属性 `focus` 定义了焦点上的 `DepthOfFieldSprite` 对象到照相机的距离。

```
DofCache.focus=2000;
```

设置 `usedof` 属性为 `true`, 就允许 `DepthOfFieldSprite` 对象显示域景深的效果, 否则 `DepthOfFieldSprite` 对象的行为就像一个普通的 `Sprit3D` 对象一样。

```
DofCache.usedof=true;  
}
```

定义在 DofCache 类里的很多属性,除 usedof 属性外,在 Camera3D 类里都有与之等效的属性,Camera3D 类的 enableDof()函数把从 Camera3D 类的这些属性复制到 DofCache 类里,并设置 usedof 属性值为 true。

这就是为什么要对 DofCache 类直接设置这些属性的值,而不用 Camera3D 类的属性的理由,这是因为 Camera3D 类的 focus 属性值经常与计算景深域的 focus 属性值不同。如果还记得,在第 7 章中设置照相机 Camera 的 focus 属性值为 2000,将减小照相机的视域,那就是我们不希望的效果,它提供了很窄的场景视口。

以后,为了得到上述效果,影响景深效果的 DofCache 类的值直接设置,而照相机 camera3D 的同样属性则保留它的空值。

在 initScene()函数里,将建立很多的 DepthOfFieldSprite 对象 sprite,并把它们添加到场景里。

```
protected override function initScene():void
{
    super.initScene();
```

在这里,定义了一个 BitmapMaterial 材质对象,并把它加到 DepthOfFieldSprite 对象的构造函数里。

也可以通过景深域精灵自己引用 BitmapMaterial 材质对象,建立 DepthOfFieldSprite 对象 sprite,其代码如下:

```
var sprite:DepthOfFieldSprite=
    new DepthOfFieldSprite(
        new BitmapMaterial(Cast.bitmap(BlackDot))
    );
```

然而,这样一来应用程序在初始时要花很多时间,给 DepthOfFieldSprite 构造函数提供一个通用的材质对象,可避免这种现象。

```
var blackDotBitmap: BitmapMaterial=new BitmapMaterial( Cast.bitmap(BlackDot));
```

建立一个保存 DepthOfFieldSprite 对象的容器 Mesh,定义它的位置,并把它放进场景。

```
container= new Mesh({z: 1000});
scene.addChild(container);
```

现在,建立 250 个 DepthOfFieldSprite 对象,并随机地把它们放在 $1000 \times 1000 \times 1000$

单位的范围里,因为这些 DepthOfFieldSprite 对象是上面建立的 Mesh 容器的子对象,所以它们的全局 Z 轴坐标的范围在 500~1500 之间,而它们的 X 坐标和 Y 坐标的范围均在 -500~500 之间。

```
var sprite:DepthOfFieldSprite;  
for(var i:int=0;i<250;++i)  
{  
    sprite=new DepthOfFieldSprite(blackDotBitmap);  
    sprite.x=Mesh.random()*1000-500;  
    sprite.y=Mesh.random()*1000-500;  
    sprite.z=Mesh.random()*1000-500  
    container.addSprite(sprite);  
}
```

在 onEnterFrame() 函数里,容器 container 围绕 Y 轴旋转,这样,这些 DepthOfFieldSprite 子对象依次旋转,修改它们到照相机的距离,展示出景深效果。

```
protected override function onEnterFrame(event:Event):void  
{  
    super.onEnterFrame(event);  
    container.rotationY += 1;  
}  
}
```

正如图 9-6 所示,那些位于景深域里的 DepthOfFieldSprite 对象,画出很清晰的黑点,当那些 DepthOfFieldSprite 对象移动得逐渐远一些,到达景深域的外面,它们就变得模糊不清。

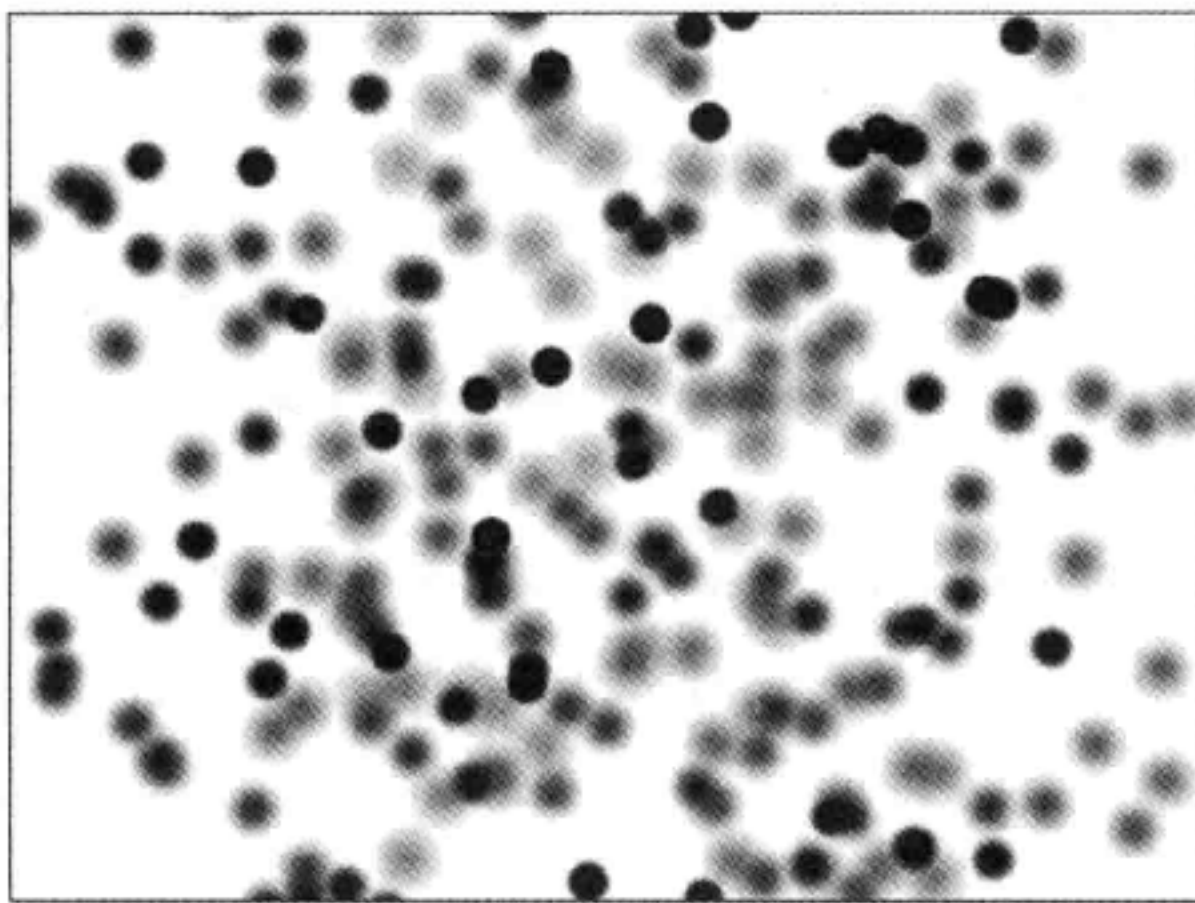


图 9-6 对象的清晰度与景深域的关系

9.4 使用粒子系统

在前面的例子里,手工建立并转换了一些 sprite 对象,在 Sprite3D 的范例里,建立了一个看起来像气球上升区域的效果,而在 DepthOfFieldSpriteDemo 范例里,建立了一个看起来像在清澈的旋转的液体里,漂浮着几百个颗粒的效果。用上面相同的方法,能建立几乎很多的各种各样最终的效果,如冒烟、炮火、雨水、烟火,但是每次逐个定义上面这些效果的属性,是很浪费时间的。而这正是粒子系统派上用途的地方。一个粒子系统提供了很多共同的类可用于很快地建立颗粒效果(如同 Away3D 里的 sprite 这样的情况)。

一般地讲,颗粒 particle 和精灵 sprite 这两个术语可以互换使用,从技术上讲,在 Away3D 与 Stardust 集成的上下文中, sprite 是一个 Away3D 的 sprite 对象,而 particle 是由 Stardust 管理的一个对象,它用于显示一个 sprite。

Away3D 并不包含 particle 系统,但是有很多 particle 系统库能与 Away3D. Flint 集成,这个库能从 <http://flintparticles.org/> 下载支持 Away3D. Flint 的 particle 系统库,包含很多范例展示它们用在 Away3D 里的方法。

Stardust 能从 <http://code.google.com/p/stardust-particle-engine/> 下载,它是另一个支持 Flash 3D 引擎的粒子库系统,在编写本书的时候,Stardust 的最新版是 1.2.163,它并不是本身就支持 Away3D,但它能很容易地把这两个库集成起来。

以下的代码使用了 Stardust version 1.2.163,它依赖 CJsymbols 库,可从 <http://code.google.com/p/cjsymbols/> 下载。

必须建立两个类来集成 Stardust 和 Away3D,第一个是初始化程序类,这个类的目的是提供一种方法,使用一些指定的参数给 Away3D 的 sprite 类的构造函数,构造出一个新的 Away3D 的 sprite 对象。实质上,这样的初始化程序类提供了一个灵活的技巧,它能在运行的时候建立新的 sprite 对象。

9.4.1 建立 Away3D Stardust 初始化程序类

在 stardust.initializers 包中,建立这个初始化类,它与 Stardust 库本身使用的格式保持一致:

```
package stardust.initializers
{
    import away3d.sprites.Sprite3D;
```

```
import idv.cjcat.stardust.common.particles.Particle;
import idv.cjcat.stardust.common.utils.construct;
import idv.cjcat.stardust.threeD.initializers.Initializer3D;
```

Away3DParticle 类初始化一个隶属于星团粒子的 Away3D sprite, 继承 Stardust Initializer3D 类, 它初始化一个 3DStardust Particle。

```
public class Away3DParticale extends Initializer3D
{
```

_constructorParams 持有要传递到 Sprite3D 的构造函数的值。

```
privat var _constructorParams: Array;
```

构造函数取出数组, 把它赋给 _constructorParams 属性。

```
public fuction Away3DPartical(constructorParams: Array=null)
{
    This. constructorParams= constructorParams;
}
```

一对 get() 和 set() 函数, 定义了允许构造函数要取回的参数和建立 Away3DPartical 对象后要设置的参数。

```
public function get constructorParams(): Array{ return
    _constrctorParams;}
public function set constructorParams(value: Array): void
{
    If (!value) value= [];
    _constrctorParams= value;
}
```

通过 Stardust 库调用 Initialize() 函数, 它把一个新的 Away3D sprite 附加到一个 particle。

```
override public function Initialize(particle: Partical): void
{
```

通过 Stardust 库提供的 construct() 函数, 可把任意数量的参数传送给指定的类, 在本例里就是 Sprite3D 类。然后这个结果的 Sprite3D 对象赋值给 Stardust particle 对象的 target 属性。这可使我们以后能从 Stardust particle 对象里, 取回对 Sprite3D 对象的引用。

```
particle.target= construct(Sprite3D, _constructorParams);
}
}
}
```

9.4.2 建立 Away3D 星团粒子渲染器

必须建立的第二个类,是粒子渲染器,这个类为星团库提供了一个在场景里添加或删除 sprite 的方法,以及把 Stardust particle 的属性转换成可视的代表粒子的 Away3D sprite 的属性。

在 stardust.renderers 包里建立一个星团粒子渲染器 Away3DParticleRenderer,再次提示,粒子渲染器与 Stardust 库本身使用的格式要保持一致。

```
package stardust.renderers
{
    import away3d.corebase.Mesh;
    import away3d.sprites.Sprite3D;
    import idv.cjcat.stardust.common.emitters.Emitter;
    import idv.cjcat.stardust.common.particles.ParticleCollection;
    import idv.cjcat.stardust.common.events.EmitterEvent;
    import idv.cjcat.stardust.common.particlesParticleIterator;
    import idv.cjcat.stardust.common.renderers.Renderer;
    import idv.cjcat.stardust.common.xml.XMLBuilder;
    import idv.cjcat.stardust.threeD.particles.Particle3D;
```

这个叫做 Away3DParticleRenderer 的类,用来渲染 Away3D 的粒子,以后再继承 Stardust Renderer 类,便可提供一个用 Stardust 库来管理粒子的功能。

```
public class Away3DParticleRenderer extends Renderer
{
```

particleContainer 属性维护着对父网格对象 Mesh 的引用,父网格保存了 Sprite3D 对象。

```
private var particleContainer:Mesh;
```

构造函数取出 Mesh 参数,把它赋给 particleContainer 属性。

```
public function Away3DParticleRenderer(particleContainer:Mesh=null)
{
    super():
    this.particleContainer= particleContainer;
}
```

render()函数用来把 Stardust particle 的属性转换到 Away3D 里代表这些粒子的 sprite 的属性。

```
protected override function render(emitter:Emitter, particles:ParticleCollection, time:Number):void
{
```

这里的循环语句遍及由 particles 参数提供的所有的粒子。

```
var particle:Particle3D;
var iter:ParticleIterator=particles.particles.getIterator();
while (particle=Particle3D(iter.particle))
{
```

如果回头看 Away3DParticle initialize() 函数,将看到把 Sprite3D 对象赋值给 Stardust particle 对象的 target 属性,在这正相反,使用 target 属性去访问 Sprite3D 对象:

```
var p:Sprite3D=particle.target as Sprite3D;
```

Stardust 的 particle3D 类,保存有自己的一套属性,这些属性定义 particle 的外貌。然而,Stardust particle 自己本身是不可见的,但它持有定义粒子应该如何显示的属性,particle renderer 类把 Stardust particle 的属性,映射到在场景里显示这些粒子的对象,在这个例子里,场景里显示这些粒子的对象就是 Sprite3D。这里,取 Stardust particle 的位置和缩放,把这些值赋给 Sprite3D 对象。

Stardust 的 particle3D 类还包含一些附加的属性,如色彩 color、蒙板 mask 和 alpha,还没有用到 Sprite3D 对象。

```
P.x=particle.x;
P.y=particle.y;
P.z=particle.z;
P.scaling=particle.scale;
Iter.next();
}
}
```

把 Stardust particle 的旋转属性,映射到 Sprite3D 对象,也应该是可能的,像如下这样:

```
P.rotation=particle.rotationZ;
```

然而,Away3D 的 3.6 版有个 bug,它导致 Sprite3D 对象不旋转。但从 SVN 库来的 Away3D 的可用版,改正了这个 bug。

把 Sprite3D 对象附加到相对父 Mesh 对象的 Stardust particle 的 particlesAdded() 函数,用 particleContainer 属性来引用。

```
protected override function particlesAdded(emitter:Emitter,particles:ParticleCollection):void
{
```



```

If (!particleContainer) return;
var particle:Particle3D;
var iter:ParticleIterator=particles.particles.getIterator();
while (particle=Particle3D(iter.particle))
{
    var p:Sprite3D=particle.target as Sprite3D;
    particleContainer.addSprite(P);
    iter.next();
}
}

```

particlesRemoved()函数,把附加到 Stardust particle 的 Sprite3D 对象从父 Mesh 对象里删除。

```

protected override function particlesRemoved(emitter:Emitter, particles:ParticleCollection):void
{
    If (!particleContainer) return;
    var particle:Particle3D;
    var iter:ParticleIterator=particles.particles.getIterator();
    while (particle=Particle3D(iter.particle))
    {
        var p:Sprite3D=particle.target as Sprite3D;
        particleContainer.removeSprite(P);
        iter.next();
    }
}

```

getXMLTagName()函数,返回这个类 Away3DParticleRenderer 的名字,这个函数由 Stardust Library 用于从一个 XML 文件载入粒子效果。

```

//XML
public override function getXMLTagName():String
{
    return "Away3DParticleRenderer";
}
//end of XML
}
}

```

9.4.3 建立星团发射器

前面已经建立了一些类,这些类允许 Away3D 使用 Stardust 包,现在可以建立一个简单的发射器 emitter,一个发射器综合了一些初始化程序,这些初始化程序定义了粒子的初始属性和动作,以及粒子如何随时被修改,Stardust 库提供了很多粒子初始属性和行为的

选项,这样使我们能只用很少的编码来建立很有趣的效果。

```
package
{
    import away3d.core.utils.Cast;
    import away3d.materials.BitmapMaterial;
    import idv.cjcat.stardust.common.actions.Age;
    import idv.cjcat.stardust.common.actions.DeathLife;
    import idv.cjcat.stardust.common.actions.ScaleCurve;
    import idv.cjcat.stardust.common.clocks.SteadyClock;
    import idv.cjcat.stardust.common.initializers.Life;
    import idv.cjcat.stardust.common.math.UniformRandom;
    import idv.cjcat.stardust.threeD.actions.Damping3D;
    import idv.cjcat.stardust.threeD.actions.Move3D;
    import idv.cjcat.stardust.threeD.actions.Spin3D;
    import idv.cjcat.stardust.threeD.emitters.Emitter3D;
    import idv.cjcat.stardust.threeD.fields.UniformField3D;
    import idv.cjcat.stardust.threeD.initializers.Omega3D;
    import idv.cjcat.stardust.threeD.initializers.Position3D;
    import idv.cjcat.stardust.threeD.initializers.Rotation3D;
    import idv.cjcat.stardust.threeD.initializers.Velocity3D;
    import idv.cjcat.stardust.threeD.zones.SinglePoint3D;
    import idv.cjcat.stardust.threeD.zones.SphereShell;
    import stardust.initializers.Away3dParticle;
```

StarDustSparksEmitter 是个 emitter 类,它继承 Stardust Emitter3D 类,这使我们能定义一个 3D 的粒子系统。

```
public class StarDustSparksEmitter extends Emitter3D
{
```

Away3d 的 Sprite3D,显示一个嵌入了 star.png 纹理图像的 BitmapMaterial 材质。

```
[Embed(source="star.png")]
protected var Star:Class;
public function StarDustSparksEmitter()
{
```

在基本的 Emitter3D 类的构造函数里,取时钟 clock 对象为参数,时钟对象定义每帧建立多少个粒子,这里用 SteadyClock 时钟类,把 0.3 传递给 SteadyClock 时钟类的构造函数参数 ticksPerCall,这样,给发射器在每帧里有 30% 的机会建立新的粒子。

```
super(new SteadyClock(0.3));
```

现在来定义粒子的初始化属性,为此用 `addInitializer()` 函数把初始化的方法传到发射器。

首先,用上面已建的初始化 `Away3DParticle` 程序,对每个新建的粒子指定对应的 `Sprite3D` 对象,把包含新 `BitmapMaterial` 材质的对象数组,传递到 `Away3DParticle` 类的构造函数,作为它的第一个参数。`BitmapMaterial` 材质对象然后会传递到由 `Away3DParticle` 类的初始函数建立的新的 `Sprite3D` 对象的构造函数。数组里的第二个参数是一个初始对象,把 `smooth` 的初始参数设为 `true`,这个参数也会传递到 `Sprite3D` 对象的构造函数,作为其第二个参数。

```
addInitializer(  
    new Away3DParticle(  
        [  
            New BitmapMaterial(Cast.bitmap(Star),  
                { smooth: true})  
        ]  
    )  
);
```

`life` 初始方法,使用 `UniformRandom` 类,设置每个新粒子的预期生命期限在 10~50 帧之间。

```
addInitializer(new Life(new UniformRandom(50,10)));
```

`Position3D` 初始方法,使用 `SinglePoint3D` 类设置每个新粒子的初始位置在 (0,0,2500)。

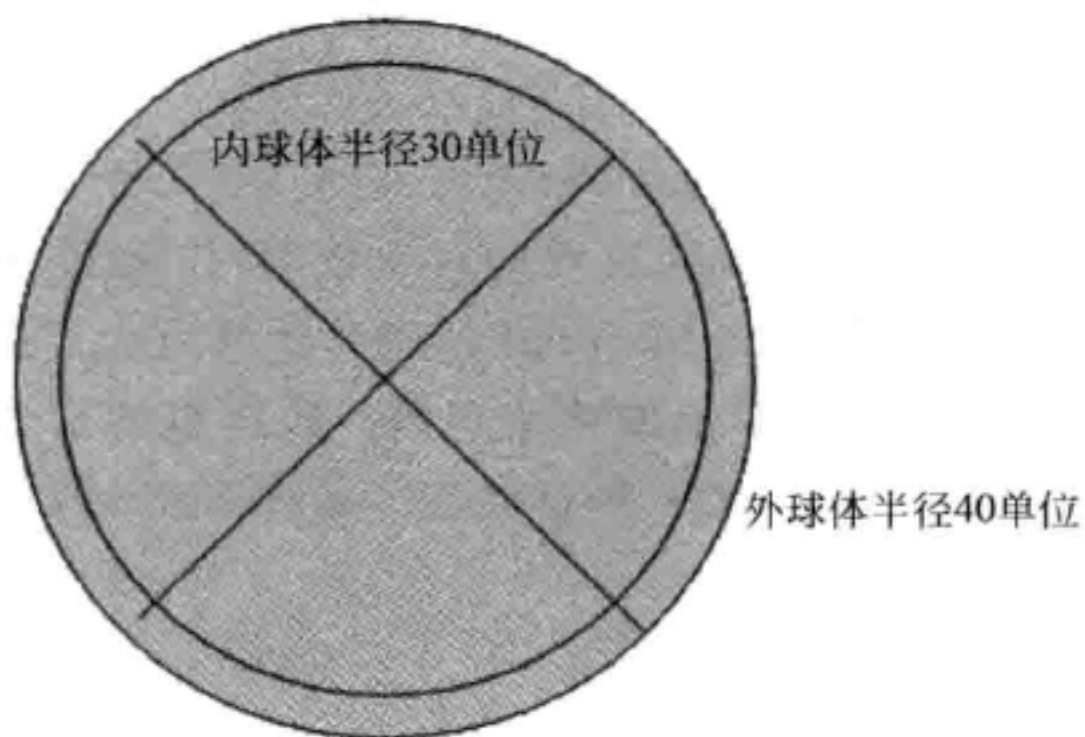
```
addInitializer(new Position3D(new SinglePoint3D(0,0,2500)));
```

`Velocity3D` 初始程序,使用 `SphereShell` 类定义每个新粒子的初始速率。速率用一个向量定义,这个向量从原点指向以原点为球心的半径范围在 30~40 间的同心球壳间的随机点上。图 9-7 显示了建立的随机向量情形(第一次,用三个参数传递到 `SphereShell` 类构造函数;第二次,用两个参数传递到 `SphereShell` 类构造函数)。

```
addInitializer(new Velocity3D(new SphereShell(0,0,0.30,40)));
```

`Rotation3D` 初始方法在这里用于设置每个新粒子围绕 Z 轴在 0° ~ 180° 之间的初始旋转角,因为 `Away3D 3.6` 版的 bug, `Sprite3D` 实际不能旋转,然而从 SVN 库下载新版 `Away3D`,并在新版的 `Away3DParticleRenderer` 的 `render()` 函数中包含旋转代码,就可实现 `Sprite3D` 旋转。

[因为 `Sprite3D` 对象总是面对照相机,围绕 X 和 Y 轴的旋转没有效果。]



从球体中心到两个球体表面之间的向量点

图 9-7 随机向量的建立

```
addInitializer(
    new Rotation3D(
        null,
        null,
        new UniformRandom(0, 180)
    )
);
```

最后,使用 Omega3D 初始化方法,设置粒子围绕 Z 轴旋转速度每帧在 $0^{\circ} \sim 5^{\circ}$ 之间。

```
addInitializer(
    new Omega3D(null, null, new UniformRandom(0, 5))
);
```

现在定义了赋给新粒子初始属性,还必须定义如何随时修改粒子,使用 addAction() 函数把新的动作类添加到发射器。

Age 的功能是减少每一个帧里每个粒子的生存时间。

```
addAction(new Age());
```

DeathLife 的功能是当粒子的 Age 到达 0 的时候,从系统中删除粒子。

```
addAction(new DeathLife());
```

Move3D 的功能是根据它在三维空间的速度,移动每帧里的粒子。

```
addAction(new Move3D());
```

Spin3D 的功能是根据它在每帧里的旋转速度,旋转粒子。

```
addAction(new Spin3D());
```


Damping3D 的功能是减少每帧里少数粒子的速度,把 0.05 提供给 Damping3D 的构造函数,我们将减少每帧里 5% 的粒子速度。

```
addAction(new Damping3D(0.05));
```

ScaleCurve 的功能是用于缩放粒子,从零增大到本来大小,然后,再回到零。因为我们把 0 提供给 ScaleCurve 构造函数的第一个参数,粒子在建立的时候,不会放大,而是本来大小。设置给 ScaleCurve 构造函数的第二个参数为 10,指定粒子在最后 10 帧里缩小到零。

```
addAction(new ScaleCurve(0,10));  
}  
}  
}
```

9.4.4 把上面的全部功能集合到一起

随着发射器的建立,现在能初始化 Away3D 引擎,Away3D 粒子渲染器和发射器本身,使用一个称为 StarDustDemo 的程序,来完成这些工作。

```
package  
{  
    import away3d.core.base.Mesh;  
    import flash.geom.Vector3D;  
    import flash.events.Event;  
    import idv.cjcat.stardust.threeD.emitters.Emitter3D;  
    import stardust.renderers.Away3DParticleRenderer;
```

建立一个新类,它继承 Away3DTemplate 类,就像迄今为止本书里所介绍的其他程序一样。

```
public class StarDustDemo extends Away3DTemplate  
{  
  
    emitter 由 emitter 属性引用。  
  
    protected var emitter:Emitter3D;  
    public function StarDustDemo()  
    {  
        super();  
    }  
    protected override function initScene():void  
    {  
        super.initScene();  
    }  
}
```

照相机的位置设置在场景原点。

```
this.camera.position=new Vector3D();
```

还必须建立一个新的发射类实例。

```
emitter=new StarDustSparksEmitter();
```

也还必须建立一个新的 Away3D 粒子渲染器类的实例。把 scene 属性传送到 Away3DParticleRenderer 的构造函数,这表示所有新建的 Sprite3D 对象直接添加到场景。

```
var renderer: Away3DParticleRenderer=  
    new Away3DParticleRenderer(this.scene);
```

然后,把 emitter 添加到 Away3D 粒子渲染器 renderer。

```
renderer.addEmitter(emitter);  
}  
protected override function onEnterFrame(event:Event):void  
{  
    super.onEnterFrame(event);
```

为了刷新粒子系统,必须调用粒子的 setup()函数,使每一帧的粒子效果得到刷新。

```
        emitter.setup();  
    }  
}
```

我们还必须做些工作,建立一些类把 Stardust 库和 Away3D 引擎结合起来,一旦完成这些就能建立一个编译过了的含有移动、旋转和缩放的粒子特效,而仅用十几行代码(如果不含 import 语句的样板代码和声明语句的函数和类),这就是 Stardust 库如此有用的原因。

StarDustDemo 程序的运行效果,如图 9-8 所示。

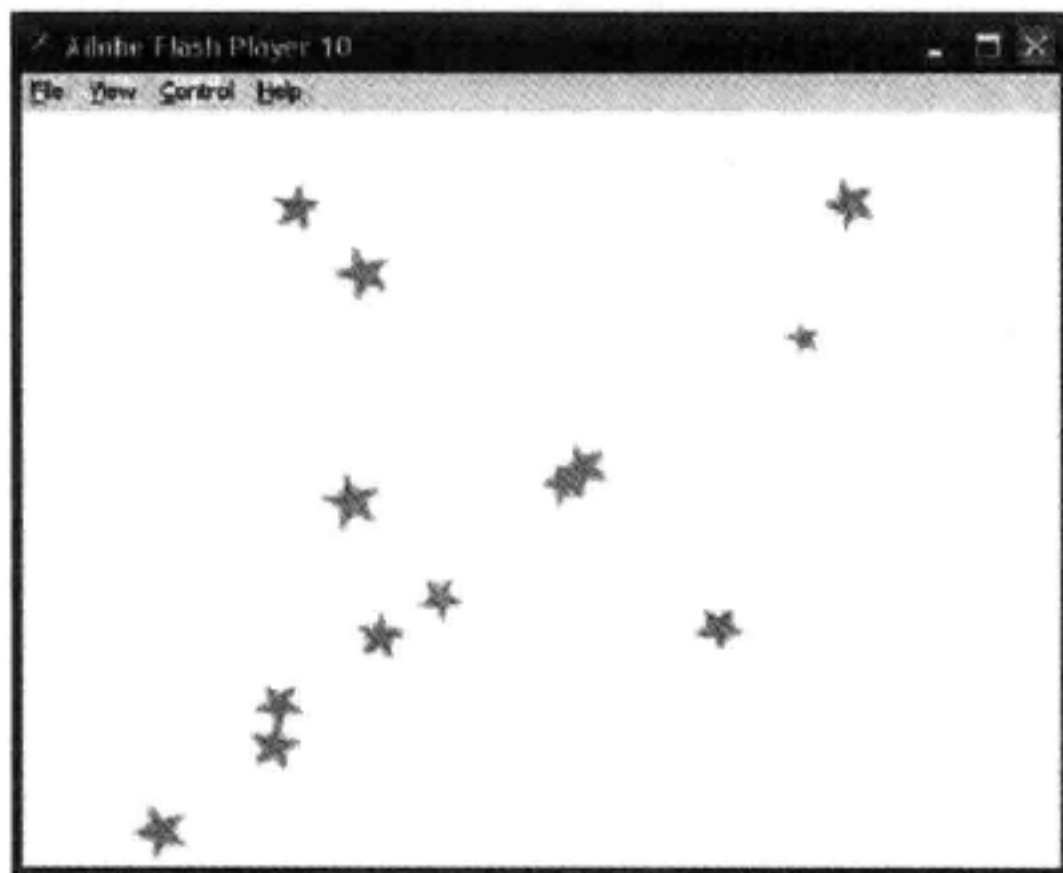


图 9-8 StarDustDemo 程序运行效果

建立3D文本

Away3D 包含很多建立 3D 对象的编程方法,在第 2 章介绍了如何从最底层开始使用基本的要素,如顶点、三角面,或用一些原始的 3D 对象类来建立 3D 对象。

最近一个时期,给 Away3D 添加了一个新的功能,使它能从文字里建立一个文字 3D 对象,这就使我们很容易地把文本加到场景里。这个功能是由一个外部库提供的,该库叫做 swfvector,它包含在 wumedia 包中。有关 swfvector 库的更多信息,可在 <http://code.google.com/p/swfvector/> 里找到,开发的这个库,还不是作为 Away3D 引擎的组成部分,但是从 2.4 到 3.4 版开始,能与 Away3D 集成,使 Away3D 能够在场景里建立并显示 3D 文本的功能。

Away3D 还能够弯曲文本对象,也就是把文字调准到由直线或曲线组成的这段轨迹。本章还介绍了一些弯曲文本对象样本的例子,以及方便调整文本的一些小窍门。

本章主要内容:

- 把文字嵌入到应用程序里
- 建立文本 3D 对象
- 把材质应用到 3D 文本
- 给 3D 文本加一些立体感
- 沿给定轨迹弯曲 3D 文本对象

10.1 嵌入文字

在 Away3D 里,建立一个文本 3D 对象时,要请求嵌入字体库的 SWF 资源文件。为适应这些要求,下面将建立一个非常简单的使用 Fronts 类的应用程序。Fronts 类嵌入了称为 Vera Sans 字体库的矢量字库 Vera.ttf 文件。

当编译完成时,产生 SWF 文件,然后,在 Away3D 里就能引用并访问嵌入的字库文件。

当使用 Flex 4 SDK 嵌入字库时,可能必须设置 embedAsCFF 属性为 false,像这样:

```
[Embed(mimeType="application/x-font", source="Vera.ttf",
fontName="Vera Sans", embedAsCFF=false)]
```

这是由于嵌入方式字库时,用的最新版 Flex SDK,有关 embedAsCFF 属性,可在网址: http://help.adobe.com/en_US/flex/using/ws2db454920e96a9e51e63e3d11c0bf6320a7fea.html 找到更多的信息。

```
package
{
    import flash.display.Sprite;
    public class Fonts extends Sprite
    {
        [Embed(mimeType="application/x-font", source="Vera.ttf",
            fontName="Vera Sans")]
        public var VeraSans:Class;
    }
}
```

这里使用的是 Bitstream Vera 字库,它是免费分发的,可从网站 <http://www.gnome.org/fonts/> 获得。然而,并不是所有的字库都是免费分发的,因此应留心一些特别的字库的版权和征税许可证的限制。

10.1.1 在场景里显示文本

文本 3D 对象,以 away3d.primitives 包里的 TextField3D 类来表示,建立文本 3D 对象需要以下两步。

(1) 解压放在独立的 SWF 文件里嵌入的字库。

(2) 建立一个新的 TextField3D 对象。

下面建立一个叫做 FontsDemo 的应用程序,它建立一个 3D 文本,然后把它添加到场景。

```
package
```

```
{
```

输入 TextField3D 类,使它在程序中是可用的。

```
import away3d.primitives.TextField3D;
```

VectorText 类用于解压嵌入在 SWF 文件里的字库。

```
import wumedia.vector.VectorText;
```

```
public class FontDemo extends Away3DTemplate
```

```
{
```

通过编译上面的 Fonts 类,建立了一个 Fonts.SWF 文件,我们希望把这个 SWF 文件作为一个原始数据嵌入进来,为此指定 MIME 的类型是 application/octet-stream。

```
[Embed(source="Fonts.swf", mimeType="application/octet-stream")]
```

```
protected var Fonts:Class;
```

```
public function FontDemo()
```

```
{
```

```
    super();
```

```
}
```

```
protected override function initEngine():void
```

```
{
```

```
    super.initEngine();
```

在建立 TextField3D 对象之前,必须解压嵌入在 SWF 文件里的字库,为此要调用 VectorText 类里的静态函数 extractFonts(),并将嵌入的 SWF 的新实例传递给它。因为我们已经指定嵌入文件 SWF 的 MIME 类型是 application/octet-stream,解压后建立的新实例类,就是一个字节数组 ByteArray。

```
VectorText.extractFont(new Fonts());
```

```
}
```

```
protected override function initScene():void
```

```
{
```

```
    super.initScene();
```

```
    this.camera.z=0;
```

下面建立一个新的 TextField3D 类的实例,它的第一个参数是字库的名字,这个名字就

是嵌入到 SWF 文件里的字库名。TextField3D 类的构造函数还要一些初始化对象参数,这些参数列在表 10-1 里。

```
var text:TextField3D=new TextField3D("Vera Sans",
{
    text:"Away3D Essentials",
    align:VectorText.CENTER,
    z:0
});
scene.addChild(text);
}
```

表 10-1 TextField3D 类构造函数的参数

参数	类型	默认值	说明
size	int	20	字的像素点数
leading	int	20	行间空格数
letterSpacing	int	0	字间空格数
text	String	""	显示的文本
width	int	500	画区的宽,如果文本大于它,回到下一行,为避免环绕,可把 TextWidth 属性值设为 number. POSITIVE_INFINITY
align	String	"TL" 或 VectorText.TOP_LEFT	定义文本的校准,VectorText 类定义了一些常数,能赋给 align 属性,这些常数是: TOP_LEFT_CENTER, TOP_LEFT, TOP_RIGHT, BOTTOM_LEFT, BOTTOM_LEFT_CENTER, BOTTOM_RIGHT, LEFT, LEFT_CENTER, RIGHT, TOP, BOTTOM 以及 CENTER

当程序运行时,场景含有一个 3D 对象,它拼写出一句话 Away3D Essentials 并使用程序指定的字体,见图 10-1。到此,文本 3D 对象能够相互转换,就像其他的对象一样。



图 10-1 TextField3D 类的应用实例效果图

10.1.2 3D 文本材质

在第2章里,如果读者还记得把 Bitmap 材质用到 3D 对象的表面是依据它们的 UV 坐标,TextField3D 对象定义的默认 UV 坐标,不允许用这样的方法使用 Bitmap 材质。然而,一些单色的材质能用于 TextField3D 对象,如 WireframeMaterial、WireColorMaterial、ColorMaterial。

10.2 突出显示 3D 文本

默认地,3D 文本对象没有立体感(虽然两面都可见到它)。一个叫做 TextExtrusion 的突显类,能用于建立另一个样子的 3D 文本对象,它用 3D 文本对象的模型,使之扩展到三维空间里。当把 TextExtrusion 和 TextField3D 对象联合起来时,能用于建立表现立体感的文本。下面的 FontExtrusionDemo 类的代码,给出了这个处理过程的例子。

```
package
{
    import away3d.containers.ObjectContainer3D;
    import away3d.extrusions.TextExtrusion;
    import away3d.primitives.TextField3D;
    import flash.events.Event;
    import wumedia.vector.VectorText;
    public class FontExtrusionDemo extends Away3DTemplate
    {
        [Embed(source="Fonts.swf", mimeType="application/octet-stream")]
        protected var Fonts:Class;
```

TextField3D 对象和 TextExtrusion 3D 对象两者都应添加为对象容器 ObjectContainer3D 的子部分,对象容器用 container 属性来引用。

```
protected var container:ObjectContainer3D;
```

text 属性,引用 TextField3D 对象,用来显示 3D 文本。

```
protected var text:TextField3D;
```

extrusion 属性,引用 TextExtrusion 对象,用来给 3D 文本添加立体感。

```
protected var extrusion:TextExtrusion;
public function FontExtrusionDemo ()
{
    super();
}
```

```

protected override function initEngine():void
{
    super.initEngine();
    this.camera.z=0;
    VectorText.extractFont(new Fonts());
}
protected override function initScene():void
{
    super.initScene();
    text=new TextField3D("Vera Sans",
    {
        text: "Away3D Essentials",
        align: VectorText.CENTER
    }
    );

```

TextExtrusion 的构造函数要引用 TextField3D 对象(或其他的网格 Mesh 对象),它也接受初始化参数,用于指定 3D 文本的立体深度,使突显的网格 Mesh 在两面都可见。

```

extrusion=new TextExtrusion(text,
{
    depth: 10,
    bothsides: true
}
);

```

建立 ObjectContainer3D 对象,提供给 TextField3D 对象和 TextExtrusion 对象,后两者为容器的子对象,ObjectContainer3D 对象的初始位置在 Z 轴正方向的 300 单元处。

```

container=new ObjectContainer3D(text, extrusion,
{
    z: 300
}
);

```

然后,把容器 container 添加到场景里,作为它的子对象。

```

scene.addChild(container);
}
protected override function onEnterFrame(event:Event):void
{
    super.onEnterFrame(event);

```

通过在每帧里修改 rotationY 属性的值,使容器缓慢地围绕它的 Y 轴旋转,在此之前的所有例子中,都会直接增加旋转属性的值,而不管这个值增加到大于 360°时的情形,最终旋转 3D 对象 180°或 540°具有完全相同的效果。然而在这样的情况下,我们希望 rotationY 的

值保存在 $0^{\circ} \sim 360^{\circ}$ 之间,以便看见旋转是否在给定的范围内,为此使用余数 `mod (%)` 运算符:

```
container.rotationY=( container.rotationY+1)%360;
```

排序 Z-sorting 的执行,是由于 `TextExtrusion` 对象和 `TextField3D` 对象是紧密地连接在一起这一事实。排序 Z-sorting 发布,导致在显示通过时它们都很明显,而有些时候,它们中的一个应该被隐藏。

为了解决这个问题,可以使用第4章里详述过的步骤,强迫改变3D对象的排序次序,这里赋给 `TextField3D` 的 `screenZOffset` 属性一个正值,强迫它当容器围绕Y轴旋转到 $90^{\circ} \sim 270^{\circ}$ 之间时, `TextField3D` 对象画在 `TextExtrusion` 对象的后面,当容器这样旋转时, `TextField3D` 对象在场景的背后。否则,赋给 `screenZOffset` 属性一个负值,强迫它画在 `TextExtrusion` 对象的前面。

```
If(container.rotationY>90 && container.rotationY<270)
    text.screenZOffset=10;
else
    text.screenZOffset=-10;
}
```

FontExtrusionDemo 程序的结果,如图 10-2 所示。

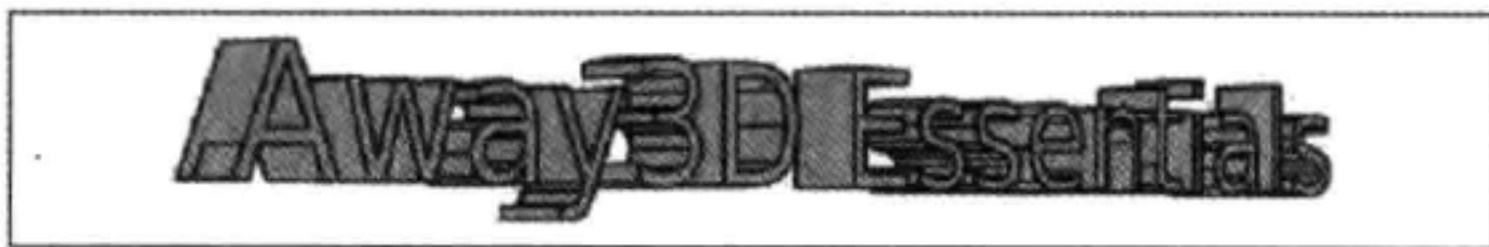


图 10-2 FontExtrusionDemo 程序运行效果

10.3 弯曲 3D 文本

Away3D 不仅能建立文本 3D 对象,还能够把文本 3D 对象校准到任意的轨迹,而个这轨迹是由直线或曲线组成的。这样就能建立一些有趣的效果,如一个文本围绕另一个 3D 对象弯曲。下面这个 `TextWarpingDemo` 类,展示如何使文本 3D 对象对齐到一条波浪形的曲线,对齐到由两条直线组成的轨迹,以及对齐到一条简单的单条曲线。

```
package
{
```

`Path` 和 `PathCommand` 类用来定义一条令文本要与之对齐的轨迹。

```
import away3d.core.geom.Path;
import away3d.core.geom.PathCommand;
import away3d.materials.ColorMaterial;
```

PathAlignModifier 类负责使 3D 对象对齐到给定的轨迹。

```
import away3d.modifiers.PathAlignModifier;
import away3d.primitives.TextField3D;
import flash.geom.Vector3D;
import flash.events.KeyboardEvent;
import wumedia.vector.VectorText;
public class TextWarpingDemo extends Away3DTemplate
{
[Embed(source="Fonts.swf", mimeType="application/octet-stream")]
protected var Fonts:Class;
protected var text:TextField3D;
public function TextWarpingDemo()
{
super();
}
protected override function initEngine():void
{
super.initEngine();
VectorText.extractFont(new Fonts());
}
protected override function initScene():void
{
super.initScene();
this.camera.z=0;
followLine();
}
protected override function initListeners():void
{
super.initListeners();
stage.addEventListener(
KeyboardEvent.KEY_UP,
onKeyUp
);
}
protected function onKeyUp(event:KeyboardEvent):void
{
switch (event.KeyCode)
{
case 49: //1
followContinuousCurve();
break;
case 50: //2
followLine();
break;
}
```

```

        case 51:           //3
        followCurve();
        break;
    }
}

```

当调用 `followContinuousCurve()` 函数、`followLine()` 函数和 `followCurve()` 函数的时候,每次都要将文本对齐到新建的轨迹, `setupText()` 函数将删除已有的 3D 文本对象(如果有),并重建一个 3D 文本对象。

```

protected function setupText():void
{
    If(text != null)
    {
        scebe.removeChild(text);
    }
}

```

读者将注意到,在希望显示的 3D 对象的字符串里,添加了几个多余的空格,这就允许文本能很好地沿着直角边对齐,这个直角是由 `followLine()` 函数里添加的两条直线组成的。

```

text=new TextField3D("Vera Sans",
{
    text: "Away3D      Essentials",
    size: 15,
    material:new ColorMaterial(0)
}
);
Scene.addChild(text);
}

```

下面的三个函数,用于把 3D 文本对象对齐到各个轨迹,第一个是 `followContinuousCurve()` 函数,它建立一个将 3D 文本对象对齐到波浪形的轨迹。

```

protected function followContinuousCurve():void
{

```

调用 `setupText()` 函数,建立一个新的 3D 文本对象。

```

setupText();

```

其次,建立一个新的 `Path` 对象,它包含很多的向量点,这些向量点定义了 3D 文本对象对齐到的轨迹。

```

var path:Path=new Path();

```

这里使用 `continuousCurve()` 函数,以 4 个点定义一条连续的曲线,4 个点为一向量数组 `Vector3D` 对象,定义曲线开始在 `(-75, -50, 300)`,向上移动到 `(-25, 50, 300)`,然后朝

下移动到(25, -50, 300),最后移动到(75, 0, 300)。

```
path.continuousCurve(
[
    new Vector3D(-75, -50, 300),
    new Vector3D(-25, 50, 300),
    new Vector3D(25, -50, 300),
    new Vector3D(75, 0, 300)
]
);
```

这些点建立的曲线,看起来如图 10-3 所示。

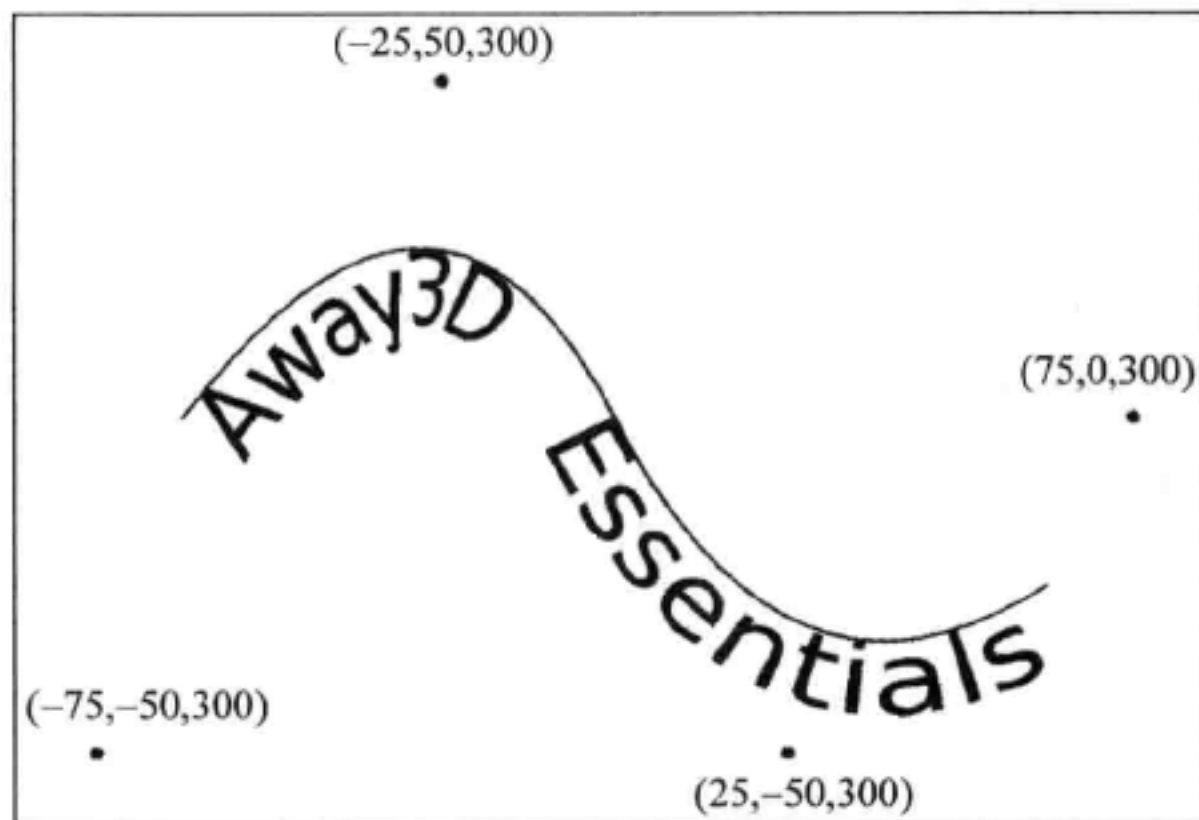


图 10-3 4 点决定的 3D 弯曲文本效果图

由于这种方法,包含计算曲线,曲线起始在第一点和第二点的中间,而结束在终点和倒数第二点之间,可在调用 `continuousCurve()` 函数的后面,添加下面的代码,来证实上述这一观点,代码如下:

```
scene.addChild(new Sphere({x: -75, Y: -50, z: 300, radius: 1, material:
newColorMaterial(0)}));
scene.addChild(new Sphere({x: -25, Y: 50, z: 300, radius: 1, material:
newColorMaterial(0)}));
scene.addChild(new Sphere({x: 25, Y: -50, z: 300, radius: 1, material:
newColorMaterial(0)}));
scene.addChild(new Sphere({x: 75, Y: 0, z: 300, radius: 1, material:
newColorMaterial(0)}));
path.debugPath(scene);
path.showAnchors=false;
```


3D 球在场景里的位置使用了我们提供给 `continuousCurve()` 函数的那些点。调用 `debugPath()` 函数时,就建立了一个 `PathDebug` 对象,它将显示曲线的轨迹,就可看到提供给 `continuousCurve()` 函数的曲线起点与终点间的那些点。

已经设置 `showAnchors` 属性为 `false`, 当它为 `true` (这是默认值) 时, `PathDebug` 对象将添加很多的 3D 球,展示组成曲线轨迹的这些点,就像我们手工完成的那样。然而,令人遗憾的是,3D 球的半径有 50 个单位,这样将填满整个场景。为解决这个问题,可以修改 `away3d.core.geom` 包里的 `PathDebug` 类文件,修改 `debug` 的 3D 球的大小,在 `PathDebug.as` 类文件的第 72 行, `addAnchor()` 函数的第一行,代码如下:

```
var sphere:Sphere=new Sphere({material:mat,
radius:50,segmentsH:2,segmentsW:20});
```

直接将 `radius` 的初始化参数 50,改为某一个值,如 5。

建立一个新的 `PathAlignModifier` 对象,用来把 3D 文本对象调整到轨迹曲线,其构造函数参数为要被调整的 3D 文本对象和调整到的轨迹曲线。

```
var aligner:PathAlignModifier=
    new PathAlignModifier(text,path);
```

然后, `execute()` 函数做出改变 3D 文本对象的请求。

```
aligner.execute();
}
```

`followLine()` 函数用于把 3D 文本对象调整到由一些直线组成轨迹曲线。

```
protected function followLine():void
{
    setupText();
    var path:Path=new Path();
```

上面用 `pass` 类的 `continuousCurve()` 函数建立了曲线轨迹,以直线建立 3D 文本对象要对准的轨迹有一点不同,对此,添加了一些 `PathCommands` 对象到 `Pass` 类的数组 `array` 属性。

提供给 `PathCommands` 结构对象的第一个参数是我们定义的命令的类型,由于定义了一条直线,所以用 `PathCommand.LINE` 常数;第二个参数是线段的起始点;第三个参数是当定义曲线时的控制点,但定义直线时,没有相应的控制点,将它设为 `null`;第四个参数是线段的起终点。

使用 `push()` 函数,把两个 `PathCommands` 结构对象推进到 `Path` 类的 `array` 属性里。

```

path.array.push(
    new PathCommand(
        PathCommand.LINE,
        new Vector3D(-75,-35,300),
        null,
        new Vector3D(-75,35,300)
    )
);
path.array.push(
    new PathCommand(
        PathCommand.LINE,
        new Vector3D(-75,35,300),
        null,
        new Vector3D(75,35,300)
    )
);

```

这些代码定义的轨迹,看起来如图 10-4 所示。

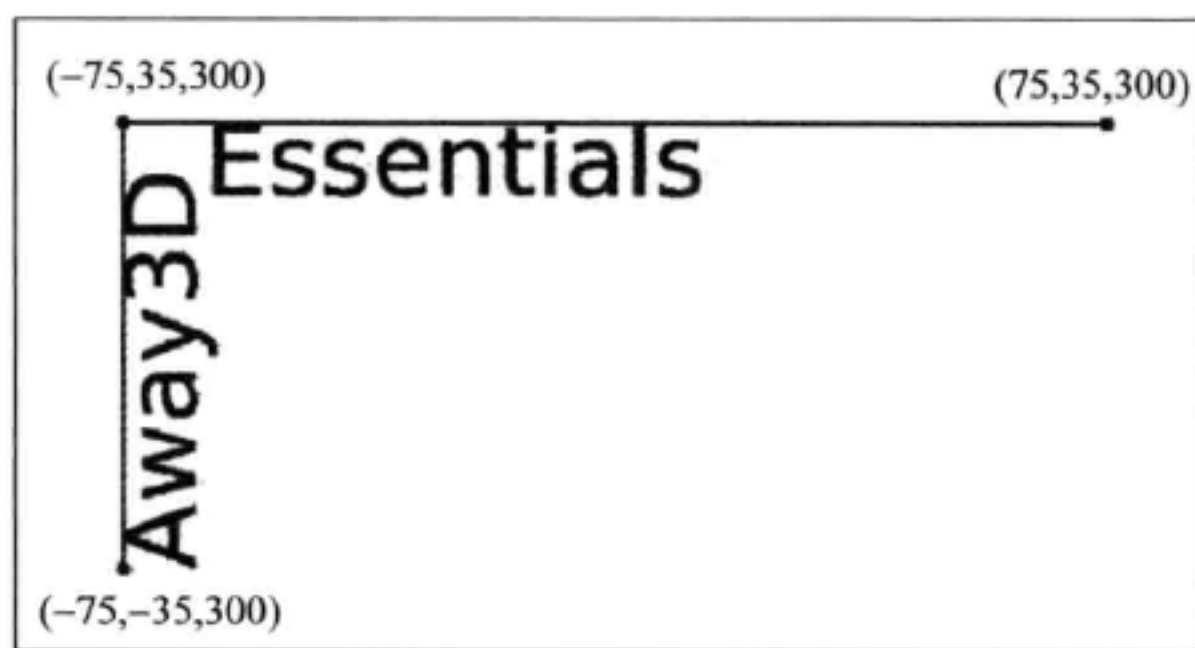


图 10-4 3 点决定的 3D 直角文本效果

尽管当在定义直线的时候,有一个修改点是毫无意义的,但会发现, PathDebug 类希望修改的点不是 null,如果用上面定义的直线轨迹,在 Path 对象上调用 debugPath() 函数时,会看到 PathDebug 对象试图读 null 的出错信息。用简单的方法可排除直线轨迹的这种错误,这就是把起点或者终点作为修改点。在下面的代码中,定义了一条与上面相同的轨迹,用了起始点和修改点是同一个点的特例。

```

path.array.push(
    new PathCommand(
        PathCommand.LINE,
        new Vector3D(-75, -35, 300),
        new Vector3D(-75, -35, 300),
        new Vector3D(-75, 35, 300)
    )
);
path.array.push(
    new PathCommand(
        PathCommand.LINE,
        new Vector3D(-75, -35, 300),
        new Vector3D(-75, -35, 300),
        new Vector3D(75, 35, 300)
    )
);

```

再一次建立一个新的 PathAlignModifier 对象,并给它 3D 文本对象和文本要调准到它的轨迹曲线 Path,然后,调用 execute()函数,做出修改。

```

var aligner:PathAlignModifier=
    new PathAlignModifier(text, path);
aligner.execute();
}

```

followCurve()函数建立一条曲线,文本要与它对齐。

```

protected function followCurve():void
{
    setupText();
    var path:Path=new Path();
}

```

就像上面建立直线轨迹一样定义曲线,也是把 PathCommand 对象添加到 Pass 类的数组 array 属性来进行的,用 PathCommand.CURVE 常数指定 PathCommand 的类型,然后指定曲线的起点、修改点和终点。

```

path.array.push(
    new PathCommand(
        PathCommand.CURVE,
        New Vector3D(-75, -45, 300),
        New Vector3D(0, 50, 300),
        New Vector3D(75, 0, 300)
    )
);

```

这将定义如图 10-5 所示的一条曲线。

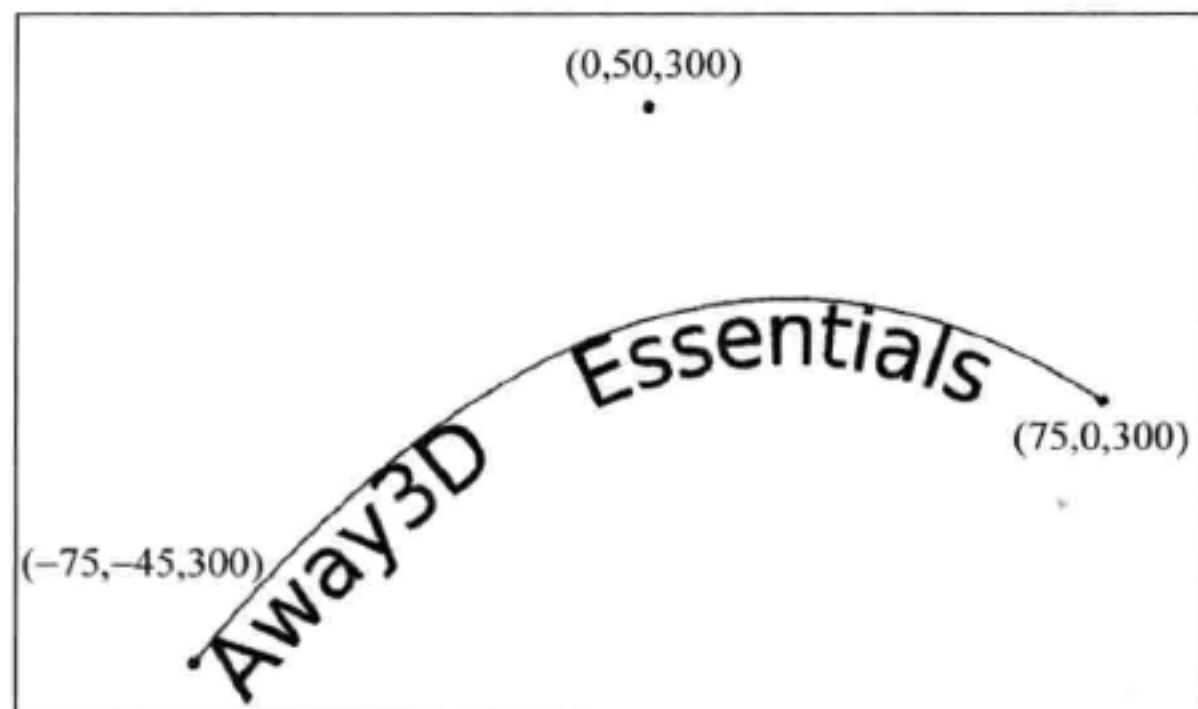


图 10-5 3 点决定的 3D 弯曲文本效果

再一次建立一个新的 PathAlignModifier 对象,并调用它的 execute() 函数。

```
var aligner:PathAlignModifier=  
    new PathAlignModifier(text,path);  
    aligner.execute();  
}  
}
```

当程序运行的时候,可按下数字 1、2 和 3 键,看到弯曲的 3D 文本对象沿三种不同的曲线弯曲的效果。

突显法和修改工具

Away3D 里有很多建立 3D 对象的方法,第 2 章涵盖了使用基本组件建立 3D 对象,同样还包含很多 `away3d.primitives` 包里的原始可用的 3D 对象。然后,在第 6 章里介绍了如何载入外部模型文件创建 3D 对象。最后,在第 10 章介绍了如何建立 3D 文本对象。

除了这些创建 3D 对象的方法之外,Away3D 还有一种方法,可创建并处理 3D 对象,这就是用 `away3d.extrusions` 和 `away3d.modifiers` 包里的类编写程序创建 3D 对象。事实上,在第 10 章中见到过这样的类,如 `TextExtrusion`,它是把平面的 3D 文本对象,变成有立体感的 3D 文本对象。

虽然用这种方法建立 3D 对象,不像使用建模软件工具那样灵活,但是能仅用少数的几行代码创建各种 3D 对象。

本章主要内容:

- 使用 `PathExtrusion` 类建立标记
- 使用 `LinearExtrusion` 类建立围墙
- 使用 `LatheExtrusion` 类建立花瓶
- 使用 `SkinExtrusion` 和 `Elevation` 建立地形图
- 使用 `ElevationRender` 类除去 3D 地形图表面的浮物

11.1 使用 `PathExtrusion` 类建立标记

`away3d.extrusions` 包里的 `PathExtrusion` 类用来沿指定路径画出一个横截面,该横截面的周边即是这条指定路径,这个横截面在英文里叫做 `profile`,它是由 `Vector3D` 对象组成

的数组。实际上,它的功能与第 10 章里介绍的 TextExtrusion 类相同,也就是给初始的平面表面添加垂直深度。但是 TextExtrusion 类把平面的 3D 文本对象,延展出与文本表面相垂直的立体,PathExtrusion 类沿 Path 对象的长度延展出一个弯曲的表面,弯曲的表面是用一系列的直线或贝塞尔(Bezier)曲线拟合成的,它与画 Path 对象的平面相垂直。这就使 PathExtrusion 类成为建立如带状、旗帜等 3D 对象的理想工具。

为展示 PathExtrusion 类的使用,下面建立一个叫做 PathExtrusionDemo 的应用程序,它建立一个简单的 3D 对象旗帜,Path 在 x 和 z 平面上,延展沿 Y 轴方向高 200 个单位。

```
package
{
    import away3d.core.gemo.Path;
    import away3d.core.utils.Cast;
    import away3d.extrusions.PathExtrusion;
    import away3d.materials.BitmapMaterial;
    import flash.gemo.Vector3D;
    public class PathExtrusionDemo extends Away3DTemplate
    {
        [Embed(source="away3dlogo.jpg")]
        protected var Away3DLogo:Class;
        public function PathExtrusionDemo()
        {
            super();
        }
        protected override function initScene():void
        {
            super.initScene();
        }
    }
}
```

为获得 3D 旗帜对象的一个良好的视口,建立一个沿 Y 轴和 Z 轴放置的照相机,然后用 lookAt() 函数,使相机能看到场景原点。

```
camera.position=new Vector3D(0,500,500);
camera.lookAt(new Vector3D(0,0,0));
```

这里建立一个新的 Path 对象,在第 10 章里,一个路径 path 对象包含很多 PathSegment 对象,依次用三个空间的位置来定义起点、控制点和终点。在这个例子里建立的 Path 对象,包含两个 PathSegment 对象,第一个 PathSegment 对象用提供给 Path 对象构造函数数组里的前三个 Vector3D 对象定义,第二个 PathSegment 对象用数组里的后三个 Vector3D 对象定义,这就会导致产生 path 对象,如图 11-1 所示。

```
var path:Path=new path(
[
    new Vector3D(-150,0,0),
    new Vector3D(-100,0,75),
```

```

new Vector3D(0,0,0),
new Vector3D(0,0,0),
new Vector3D(100,0,-75),
new Vector3D(150,0,0)
]
);

```

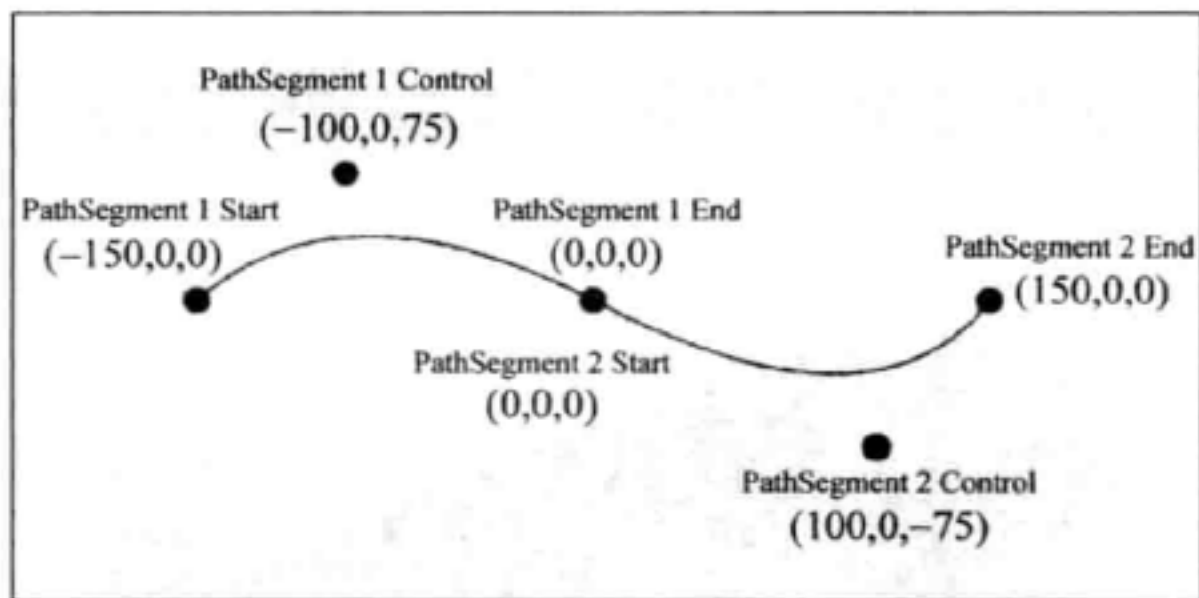


图 11-1 3D 旗帜的横向横截面

与 Path 对象所处平面垂直的横截面叫做 profile, 定义为一个 Vector3D 对象数组, 在本例里, 横截面是一条高度为 200 单元的线条。

```

var profile:Array=
[
    new Vector3D(0,-100,0),
    new Vector3D(0,100,0)
];

```

然后, 用 Path 对象和横截面 profile 对象创建 PathExtrusion 对象, subdivision 初始化对象参数用来定义导出 3D 对象的细节, 较高的值构成的最后的 3D 对象使用较多的三角面产生比较光滑的显示。

```

var extrusion:PathExtrusion=new PathExtrusion(
    path,
    profile,
    null,
    null,
    {
        material: new BitmapMaterial(Cast.bitmap(Away3DLogo)),
        bothsides: true,
        subdivision: 10
    }
);

```

然后,把创建的 PathExtrusion 对象添加到场景。

```
        scene.addChild(extrusion);  
    }  
}  
}
```

图 11-2 显示了应用程序的输出。

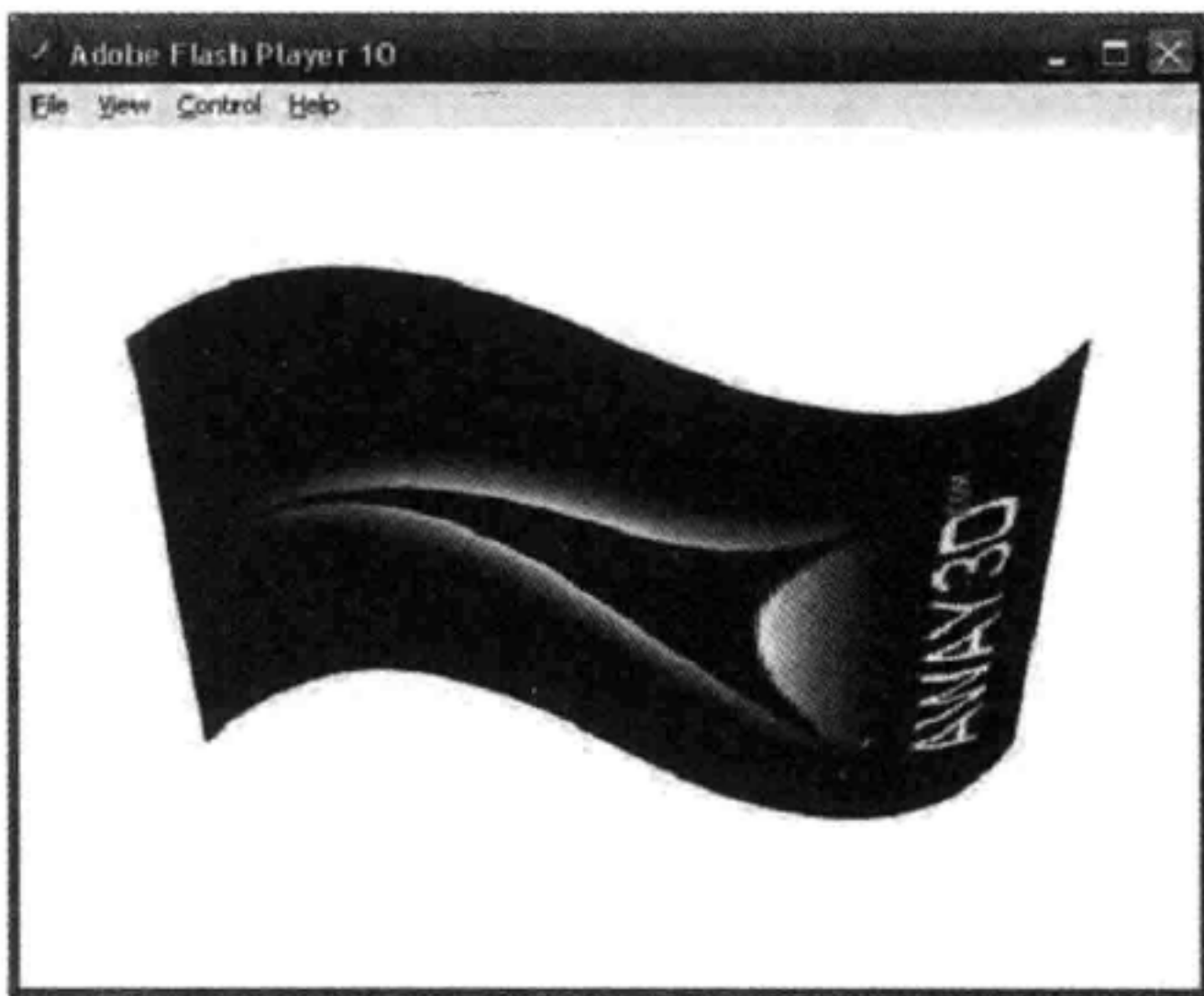


图 11-2 PathExtrusionDemo 程序运行效果

11.2 使用 LinearExtrusion 类建立围墙

LinearExtrusion 类用于建立一个立体的矩形围墙,这些矩形围墙的横截面是由一条或多条直线组成的,这对于建立由多条直线组成的围墙是非常有用的,如建筑的平面结构图。在下面的 LinearExtrusionDemo 应用程序里,使用 LinearExtrusion 类,创建代表 L 形房子的围墙 3D 对象。

```
package  
{  
    import away3d.extrusions.LinearExtrusion;  
    import flash.events.Event;  
    import flash.geom.Vector3D;  
    public class LinearExtrusionDemo extends Away3DTemplate
```



```

{
protected var walls:LinearExtrusion;
public function LinearExtrusionDemo()
{
    super();
}
protected override function initScene():void
{
    super.initScene();
    camera.position=new Vector3D(1000,750,1000);
    camera.lookAt(new Vector3D(0,0,0));
}

```

wallPoints 数组是一些点的集合,这些点定义了多条连接的线,这些线定义了基地轮廓的围墙,用矩形 3D 对象组成这些围墙,前条线的终点是下条线的起点,这些点用下面的代码定义一个横截面,如图 11-3 所示。

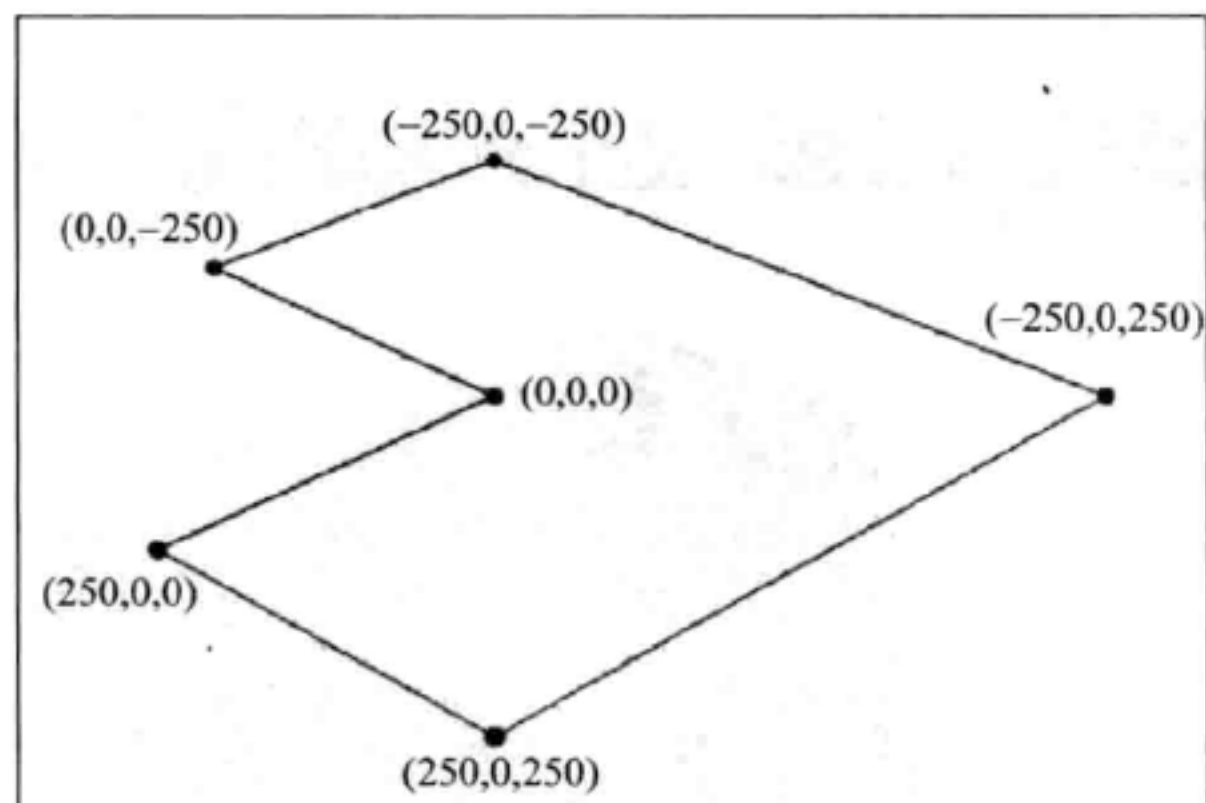


图 11-3 矩形围墙的横向横截面

```

var wallPoints:Array=
[
    new Vector3D(-250,0,-250),
    new Vector3D(0,0,-250),
    new Vector3D(0,0,0),
    new Vector3D(250,0,0),
    new Vector3D(250,0,250),
    new Vector3D(-250,0,250),
    new Vector3D(-250,0,-250)
];

```

这里定义一个新的 LinearExtrusion 对象,Offset 初始化对象参数,用来定义围墙的高

度,thickness 参数定义其宽度,recenter 参数用来确保导出的 3D 对象位于它的中间。

```
walls=new LinearExtrusion(  
    wallPoints,  
    {  
        thickness:10,  
        offset: 150,  
        recenter: true  
    }  
);  
scene.addChild(walls);  
}  
}
```

图 11-4 显示了这个应用程序的输出。

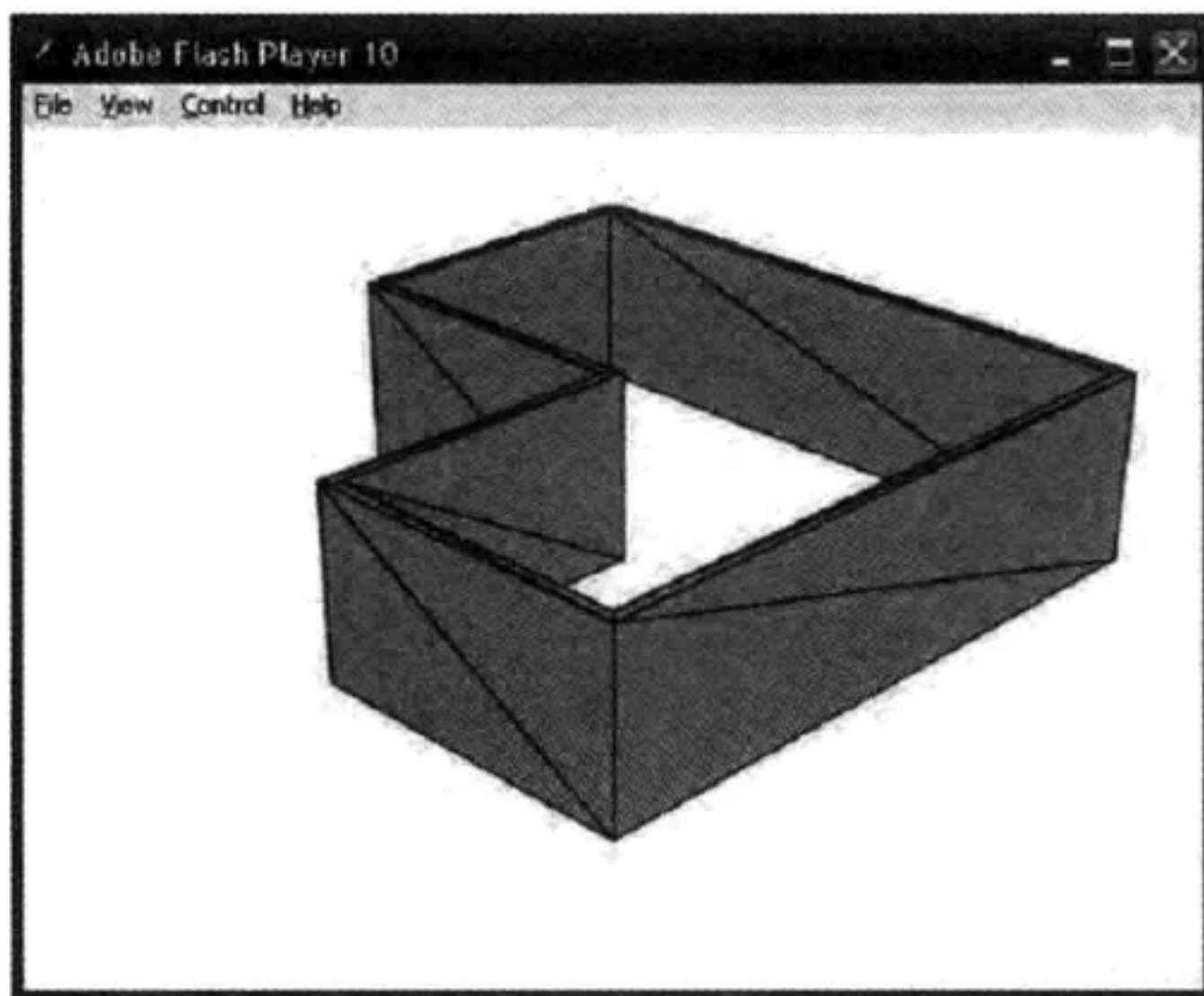


图 11-4 LinearExtrusionDemo 程序运行效果

11.3 使用 LatheExtrusion 类建立花瓶

木匠使用一种叫车床的机器,将木块旋转进行雕刻。车床加工能用于建立一些对象,如碗、大酒杯、花瓶或其他一些对象,只要它们围绕中心轴旋转时,这些对象的横截面不变即

可。这样的对象叫做轴对称。

Away3D 里用 LatheExtrusion 类建立上述对象,这些对象用一个横截面并且围绕轴(默认是 Y 轴)旋转,产生立体的 3D 对象。下面的 LatheExtrusionDemo 应用程序使用 LatheExtrusion 类创建花瓶的 3D 对象。

```
package
{
    import away3d.extrusions.LatheExtrusion;
    import flash.gemo.Vector3D;
    public class LatheExtrusionDemo extends Away3DTemplate
    {
        protected var vase:LatheExtrusion;
        public function LatheExtrusionDemo()
        {
            super();
        }
        protected override function initScene():void
        {
            super.initScene();
            camera.position=new Vector3D(0,500,500);
            camera.lookAt(new Vector3D(0,0,0));
        }
    }
}
```

图 11-5 定义的横截面将传递给 LatheExtrusion 类,读者可能已经注意到了,组成横截面的 X 坐标点,其坐标值都是正值,并且没有跨过 Y 轴(请记住,Y 轴是横截面旋转时围绕的默认旋转轴),这确保横截面旋转时,不会自我相交。这是一个好办法,它确保提供给 LatheExtrusion 类的横截面不会跨过它旋转时的中心轴。

正像使用 LinearExtrusion 类一样,横截面定义为一个 Vector3D 对象的数组。

```
var profile:Array=[
    new Vector3D(50,200,0),
    new Vector3D(40,150,0),
    new Vector3D(60,120,0),
    new Vector3D(40,0,0)
];
```

然后,新建一个 LatheExtrusion 类的实例,centerMesh 初始化对象参数的使用方法,几乎与 LinearExtrusion1 类里的 recenter 初始化对象参数相同。

必须设置 flip 初始化对象参数为 true,使三角形朝向导出的 3D 对象的表面,以便从照

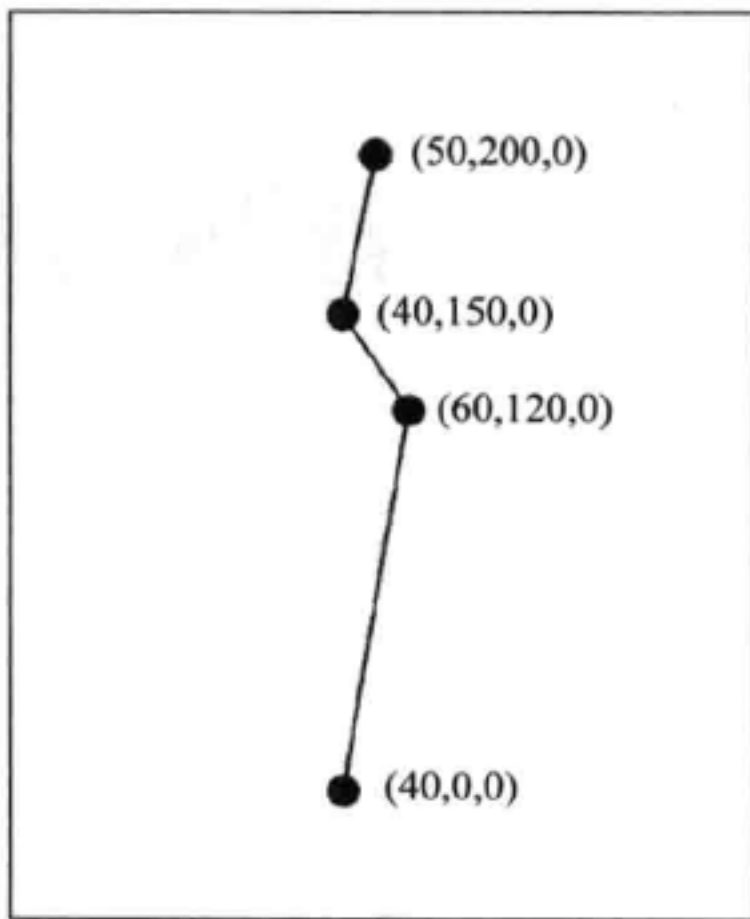


图 11-5 花瓶的纵向横截面

相机的视点可见到它们。如果使 flip 保留其默认值 false, 将直接看过去, 见到花瓶的边缘的外面。

LatheExtrusion 类的结构函数编码文档, 实际上, 将 recenter 列为一个有效的初始化对象参数, 这是不对的, LatheExtrusion 类的结构函数不会处理 recenter 初始化对象参数。

```
vase=new LatheExtrusion(  
    profile,  
    {  
        subdivision: 12,  
        centerMesh:true,  
        thikness:10,  
        flip:true  
    }  
);  
scene.addChild(vase);  
}  
}
```

图 11-6 显示了这个应用程序的输出。



图 11-6 LatheExtrusionDemo 程序运行效果

11.4 使用 SkinExtrusion 类建立地形图

不像上面所见的 extrusion 类, SkinExtrusion 类并不因为建立附加的维数而延展出横截面, 相反, 而是取出一系列阵列点, 并用它们定义为能添加到场景里的 3D 对象的顶点。表面上看, 使用 SkinExtrusion 类, 并没有很多好处, 不如第 2 章里用手工创建网格 Mesh 对象。然而, 在实践上, SkinExtrusion 类使用的列阵列点, 是由另一个叫做海拔的 Elevation 类产生的。然后, 海拔类 Elevation 取出一个图形 image, 叫做海拔图 height map, 海拔类依海拔图上的各个阵列点的颜色深浅, 在平面上面画出各个阵列点的高度。

在图 11-7 里, 可以看见上面所述的工作。黑白平面是具有同一个海拔图的平面, 它用来画出各个红点的高度, 由图可见, 海拔图上较亮的区域上的这些点, 占据较高的位置。

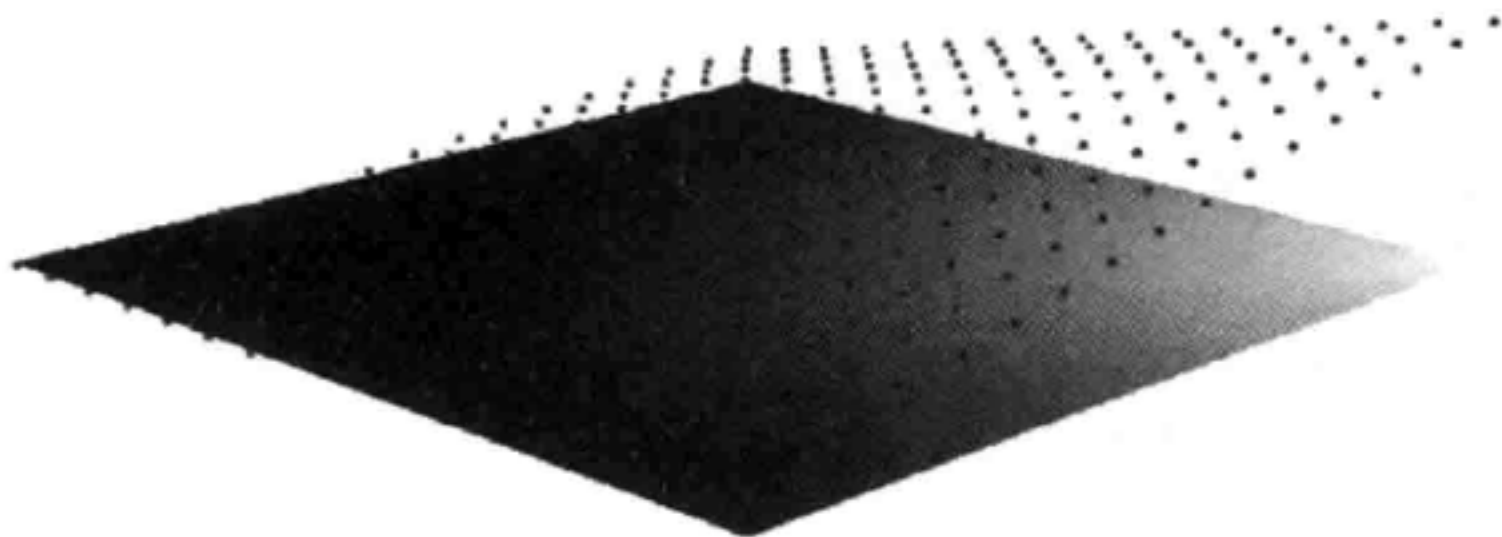


图 11-7 海拔图的高度示意图

在 SkinExtrusionDemo 的应用程序中, 用海拔图(图 11-8 左图)和纹理图(图 11-8 右图)建立一个户外地形。

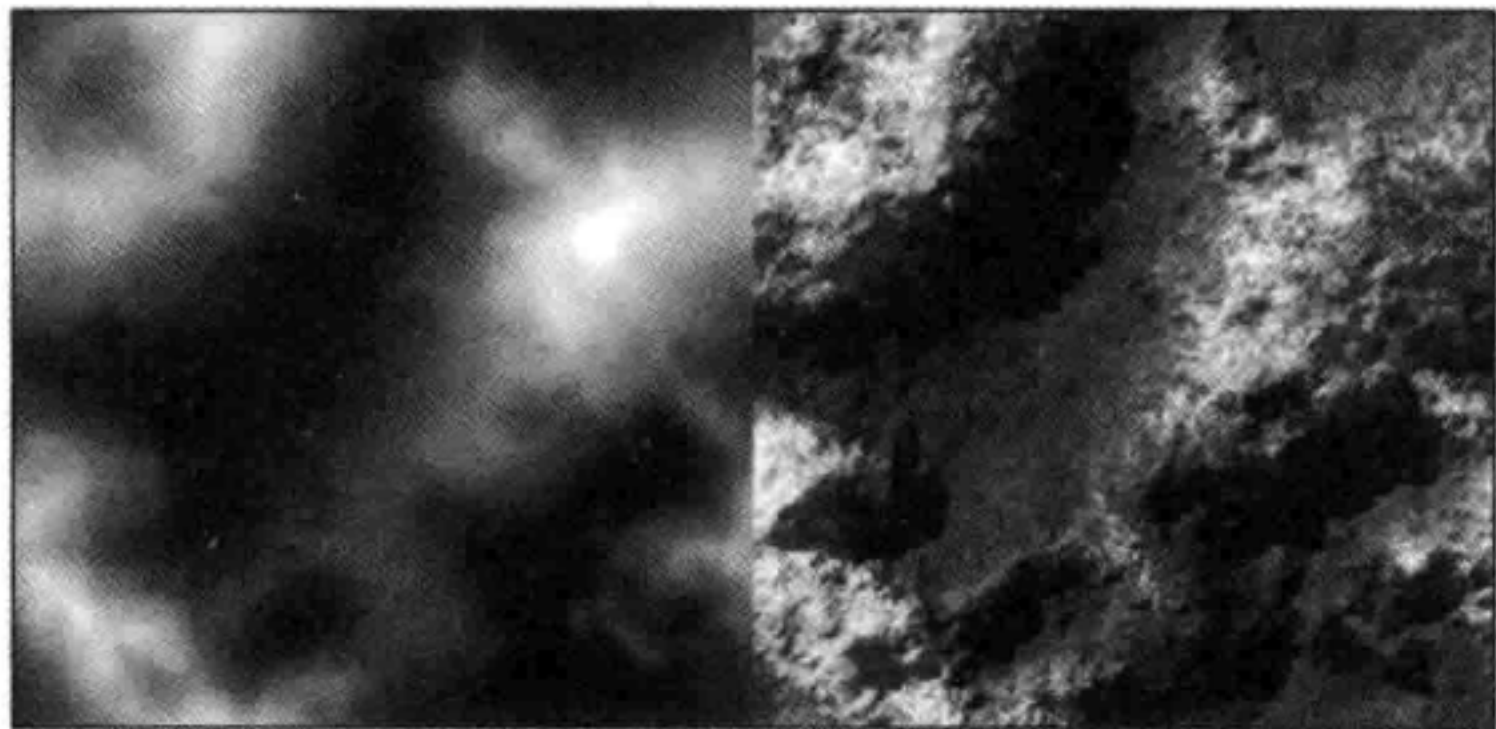


图 11-8 SkinExtrusionDemo 程序的海拔图和纹理图

也可用免费的 T2 应用程序 <http://www.toymaker.info/html/texgen.html> 建立一个逼真有纹理的海拔图。

```
package
{
    import away3d.core.utils.Cast;
    import away3d.extrusions.Elevaion;
    import away3d.extrusions.SkinExtrude;
    import away3d.materials.BitmapMaterial;
    import flash.gemo.Vector3D;
    public class SkinExtrusionDemo extends Away3DTemplate
    {
```

为方便访问,把海拔图和纹理图嵌入进来:

```
[Embed(source="heightmap.jpg")]
protected var Heightmap:Class;
[Embed(source="terrain.jpg")]
protected var Terrain:Class;
protected var extrude:SkinExtrude;
public function SkinExtrusionDemo()
{
    super();
}
protected override function initScene():void
{
    super.initScene();
    camera.position=new Vector3D(400,200,400);
    camera.lookAt(new Vector3D(0,0,0));
    var terrainMaterial:BitmapMaterial=
        new BitmapMaterial(Cast.bitmap(Terrain));
```

在能使用 SkinExtrusion 类之前,还必须建立海拔类 Elevation 的实例:

```
var elevation:Elevation=new Elevation();
```

然后,使用 generate() 函数,建立一个 Vector3D 对象的多维数组,这个数组是 SkinExtrusion 类以后要用的。传递给 generate() 函数的第二个参数,与它的默认值相同。

第一个参数指定了表示海拔图的 BitmapData 对象。

第二个参数定义了彩色通道,这个通道是用于读出确定导出点高度数据的,字符串“r”表示读像素点的红色。

因为海拔图是灰度图形,它上面的任何像素点,其红色、绿色和蓝色值都是相同的,这就使得无论 `generate()` 函数使用什么彩色通道,都是一样的。

第三个和第四个参数定义了沿海拔图的宽度和沿海拔图的高度各读出多少个样本。较高的值使导出的 3D 对象越精细。

第五个和第六个参数定义了平面的纵横刻度,导出点安放在纵横交点上。默认值是 1,表示平面的纵横刻度与海拔图是一样的。

第七个参数,没有使用它的默认值,它用来确定地形图高度的比例尺,按默认的划分,地形图的最大高度为 255, `generate()` 函数的默认值是 1,因为我们提供的比例值是 0.25,所以,在 3D 对象地形图里最大高度是默认值的 1/4。

```
var vertices:Array=elevation.generate(  
    Cast.bitmap(Heightmap),  
    "r",  
    16,  
    16,  
    1,  
    1,  
    0.25  
);
```

现在使用这个数组产生一个新的 `SkinExtrude` 3D 对象。 `coverall` 初始化对象参数,设置为 `true`,确保用到 3D 对象的材质覆盖整个导出 3D 对象。如同 `LinearExtrusion` 类一样, `recenter` 参数将调整组成 3D 对象的顶点位置,把 3D 对象中心调整到网格的原点位置。在调整的过程中,也把 3D 对象的高度考虑在内。这就表示,当 `recenter` 参数为 `true` 的时候,3D 对象的最高点将在 3D 对象本地坐标原点的上面,而最低点将低于它。在图 11-9 里,可看见调整过的 `SkinExtrude` 3D 对象的本地坐标的原点,以黑色圆点的位置表示。

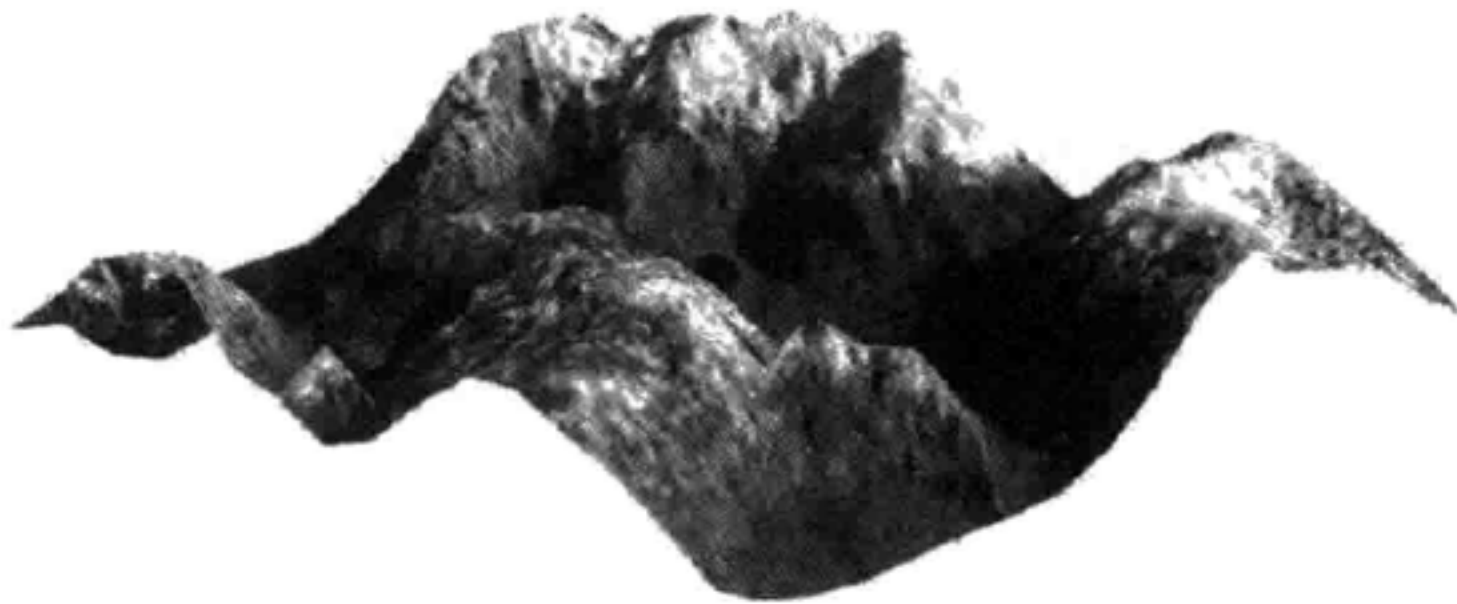


图 11-9 SkinExtrusionDemo 程序的运行效果一

如果 `recenter` 参数为 `false`, 也就是它的默认值, 网格原点的位置将在一个角落, 它所有的顶点在原点的上面。图 11-10 中的黑色圆点显示 `SkinExtrude 3D` 对象的本地坐标原点位置, 此时的 `SkinExtrude 3D` 对象还没有回到中心位置。

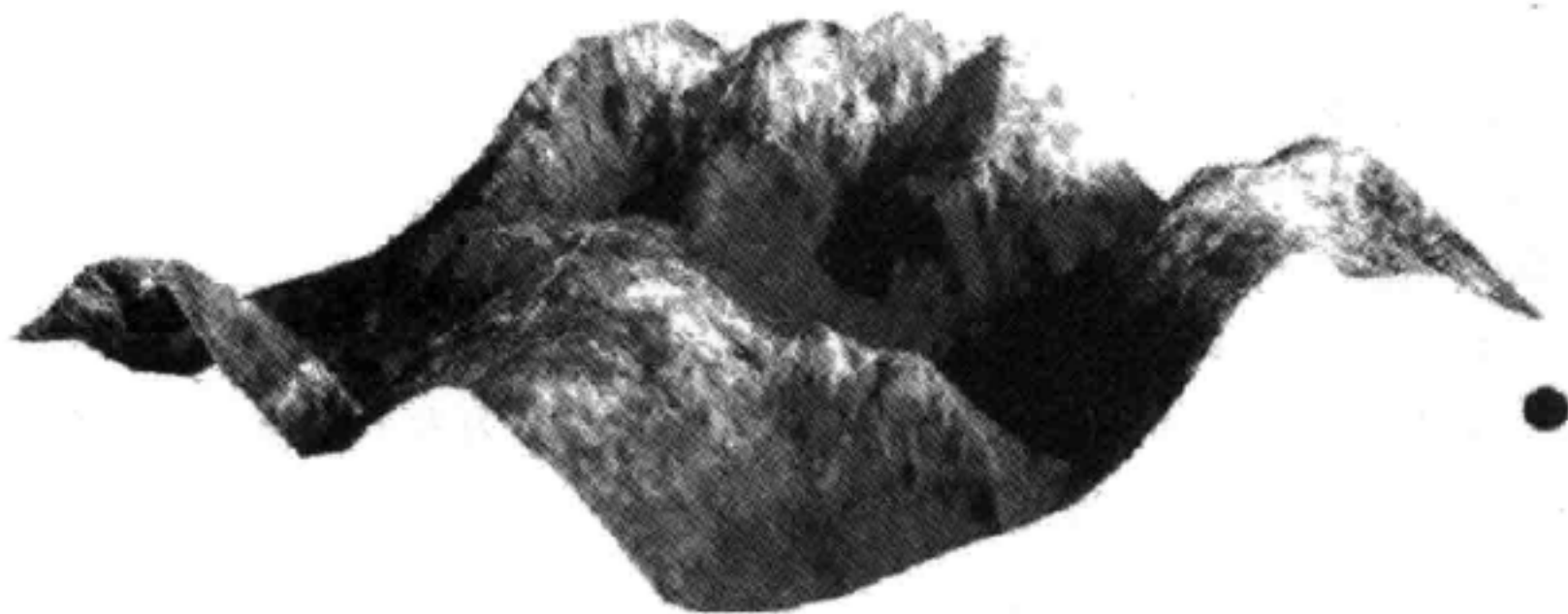


图 11-10 `SkinExtrusionDemo` 程序的运行效果二

```
extrude=new SkinExtrude(vertices,  
{  
    coverall:true,  
    material:terrainMaterial,  
    recenter: true,  
    bothsides:true  
}  
);
```

`generate()` 函数返回凡是其边缘与 X、Y 轴平行的点, 在平面上方的位置。当这些点传送到 `SkinExtrude` 类的时候, 将能有效地建立一个 3D 对象, 从 X 轴的正向看 Y、Z 平面, 它像一个崎岖不平的墙, 通过围绕 X 轴旋转 90° , 朝它看时它像是表示地面。

```
extrude.rotationX=90;
```

设置 `recenter` 属性为 `true`, 将使 3D 对象复位。这里直接把它的位置设置到场景的原点。

```
extrude.x= extrude.y= extrude.z=0;  
scene.addChild(extrude);  
}  
}  
}
```

平面截图显示程序运行的结果如图 11-11 所示。

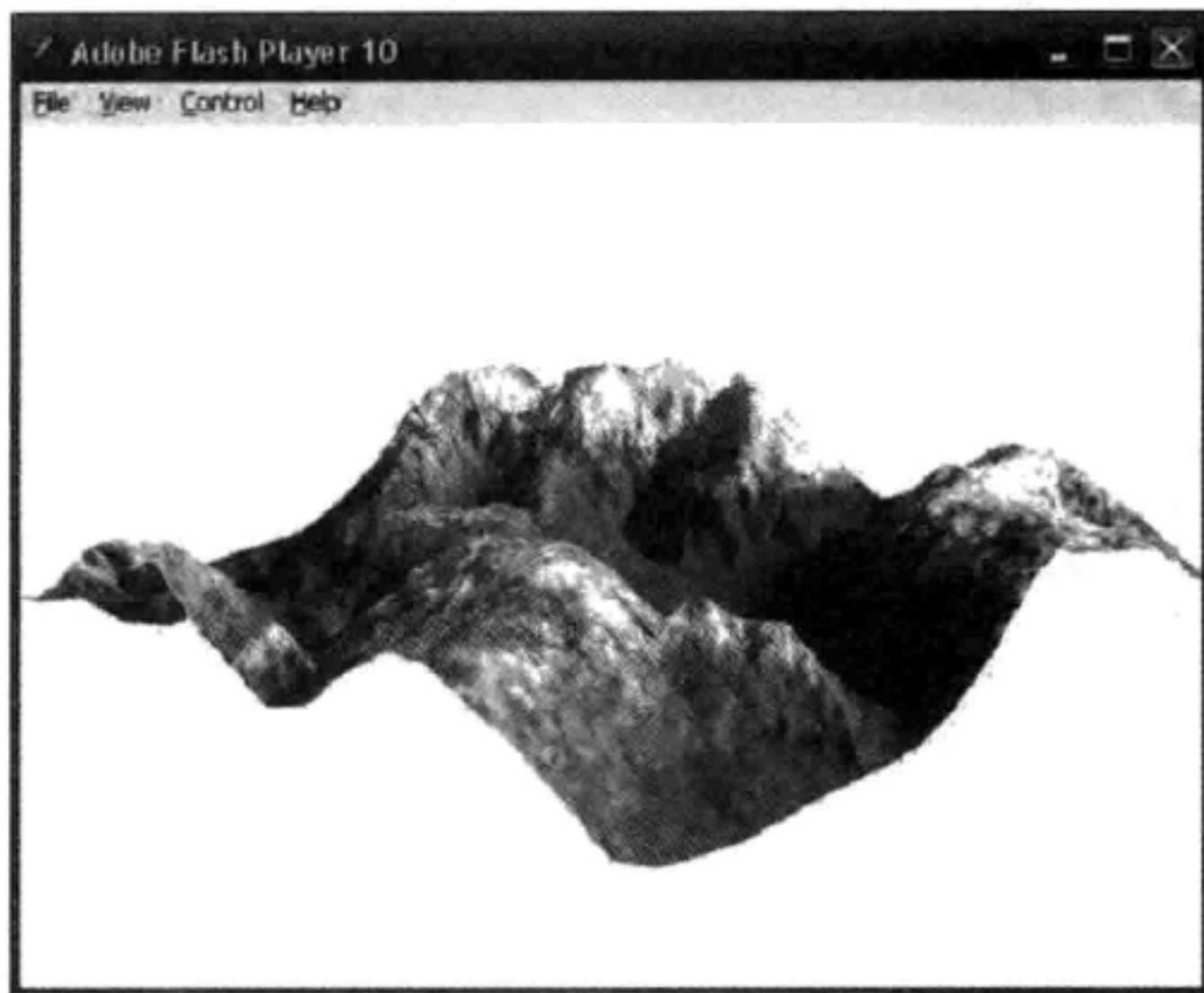


图 11-11 SkinExtrusionDemo 程序的运行效果三

11.5 用海拔阅读类读出地形图表面高度

当使用由 SkinExtrusion 类建立的 3D 对象的时候,在 3D 对象的表面上给定一个点,需要获得它的高度,这是常有的事。例如,模拟驾驶小轿车穿越地形的情形,为了确保轿车的安全,车的位置必须高于用 SkinExtrusion 建立的 3D 地形对象表面高度。ElevationReader 类就能用于这种用途。当创建一个海拔对象 Elevation 的时候,给它提供一个海拔图,海拔对象 Elevation 的 generation() 方法,返回海拔图上的各个点, ElevationReader 类的 getLevel() 函数就能得到各个相应点的高度。这个值即可用于与 SkinExtrusion 建的 3D 对象对应的位置的高度。

下面的 ElevationReaderDmo 应用程序,共享了 SkinExtrusionDemo 应用程序中的一些代码,另外添加了一些创建 ElevationReader 类的代码,这些代码用于当一个球随机地在 SkinExtrusion 3D 对象的表面上运动的时候,修改球的高度。

```
package
{
    import away3d.core.utils.Cast;
    import away3d.extrusions.Elevaion;
    import away3d.extrusions.ElevaionReader;
    import away3d.extrusions.SkinExtrude;
    import away3d.materials.BitmapMaterial;
```

```

import away3d.primitives.Sphere;
import com.greensock.TweenLite;
import flash.events.Event;
import flash.gemo.Vector3D;
public class ElevationReaderDemo extends Away3DTemplate
{
    [Embed(source="heightmap.jpg")]
    protected var Heightmap:Class;
    [Embed(source="terrain.jpg")]
    protected var Terrain:Class;
    protected var extrude:SkinExtrude;
    protected var sphere:Sphere;
    protected var elevationreader:ElevationReader;
    public function ElevationReaderDemo()
    {
        super();
    }
    protected override function initScene():void
    {
        super.initScene();
        camera.position=new Vector3D(400,200,400);
        camera.lookAt(new Vector3D(0,0,0));
        var terrainMaterial:BitmapMaterial=
            new BitmapMaterial(Cast.bitmap(Terrain));
        var elevation:Elevation=new Elevation();
        var verticies:Array=elevation.generate(
            Cast.bitmap(Heightmap),
            "r",
            16,
            16,
            1,
            1,
            0.25
        );
        extrude=new SkinExtrude(verticiees,
        {
            coverall:true,
            material:terrainMaterial,
            recenter: true,
            bothsides:true
        }
        );
        extrude.rotationX=90;
        extrude.x= extrude.y= extrude.z=0;
        scene.addChild(extrude);
    }
}

```

创建 ElevationReader 对象的过程,与创建 Elevation 对象的过程非常相似。首先要初始化一个新 ElevationReader 对象的实例:

```
elevationreader=new ElevationReader();
```

然后,调用 traceLevels()函数,给它提供的参数要与给 Elevation 的 generation()函数的参数一样准确,这一点是很重要的,也就是传递给 generation()函数和 traceLevels()函数的参数是相同的,因为要确保 getLevels()函数下的返回值与 SkinExtrude 3D 对象的高度一致。

```
elevationreader.traceLevels(  
    Cast.bitmap(Heightmap),  
    "r",  
    16,  
    16,  
    1,  
    1,  
    0.25  
);
```

然后建立一个球体,它随机地穿越地形表面运动。

```
sphere=new Sphere({radius:10});  
scene.addChild(sphere);
```

最后,调用 moveSphere()函数,开始球的缓动操作。

```
moveSphere();  
}  
override protected function onEnterFrame(event:Event):void  
{  
    super.onEnterFrame(event);
```

调整每帧里球的高度与地形高度相匹配,这由调用 getLevels()函数来确定。

给 getLevels()函数提供球沿 X 轴和 Z 轴的位置坐标值,作为首要的两个参数,而当我们想到这两个参数实际上是调用球 X 轴和 Y 轴的坐标值的时候,这似乎混淆了。把球在的 Z 轴上的位置提供为 Y 轴参数的理由是:因为 SkinExtrude 对象围绕 X 轴旋转了 90°,不幸的是,ElevationReader 类没有相同的旋转功能。这就表示必须手工地把球沿世界空间里的 Z 轴运动,转换到 ElevationReader 类本地坐标里沿 Y 轴的垂直运动。

第三个参数用于提供给由 getLevels()返回高度的抵消量。

回想创建 SkinExtrude 3D 对象的代码,会注意到 recenter 初始化对象参数是设置为 true,这会产生如此的效果:使 SkinExtrude 3D 对象的本地原点,移到了它的场景中心,与它的一个角相对应。这也就表示,SkinExtrude 3D 对象的最低点(即相应海拔图上最黑区里的那些点)将移到中心点的下面,而最高点在中心点的上面。

在 SkinExtrude 3D 对象对准它的场景中心时,提供给 getLevels()函数的本地 x 和 y 参数,不必调整为假想中心的值,然而, getLevels()函数的返回值总是正值,这是因为 getLevels()函数没有把 SkinExtrude 3D 相应的高度的改变考虑在内。这就是为什么在计算位移量时,首先减去 $255 \times 0.25 \times 0.5$ 的原因,如此,就调整了 getLevels()函数的返回值,使其与 SkinExtrude 3D 对象(请记住, SkinExtrude 3D 对象的最大高度是 255 单元,以 0.25 比例缩小,然后中心化,又下降一半)的实际最小高度相匹配,然后,把球的半径加进去,确保球的底部在 SkinExtrude 3D 对象的上面。

如果将 SkinExtrude recenter 初始化对象参数设置为 false,必须调整提供给 getLevels()函数的 x 和 y 参数(128,是地形宽度和深度的一半值),像这样:

```
sphere.y=elevationreader.getLevel(
sphere.x+128,
sphere.z+128,
sphere.radius
);
```

```
sphere.y=elevationreader.getLevel(
sphere.x,
-sphere.z,
-255*0.25*0.5+sphere.radius
);
}
```

moveSphere()函数用于建立递归的缓动操作,这使运动球随机地确定在地形上的位置。

```
protected function moveSphere():void
{
TweemLite.to(sphere,2,
x: Math.random()*256-128,
z: Math.random()*256-128,
```

通过设置 onComplete 参数,递归地调用 moveSphere()函数,能确保球连续地运动到随机点。

```
onComplete:moveSphere
}
);
}
}
```

当应用程序运行时,可以看见,球穿越地形表面运动,而它总保持在地形的上面,如

图 11-12 所示。



图 11-12 球体穿越地形表面的运动

11.6 高度映射修改类

我们已经看到使用海拔图修改 SkinExtrusion 或 Elevation 类的表面平面的高度,使用 HeightMapModifier 类是同样的原理,但是有限度地修改表面高度,它能用于任何平面的、立体的、球面的和用外部模型文件载入的复杂的 3D 对象的表面高度。为此, HeightMapModifier 类是通过修改 3D 对象的表面的法向矢量的位置来实现的(法向矢量就是一个垂直于表面的单位矢量)。

以下的 HeightMapModifierDemo 应用程序,将把如图 11-13 所示海拔图,用于代表月亮或其他小行星的球面上。

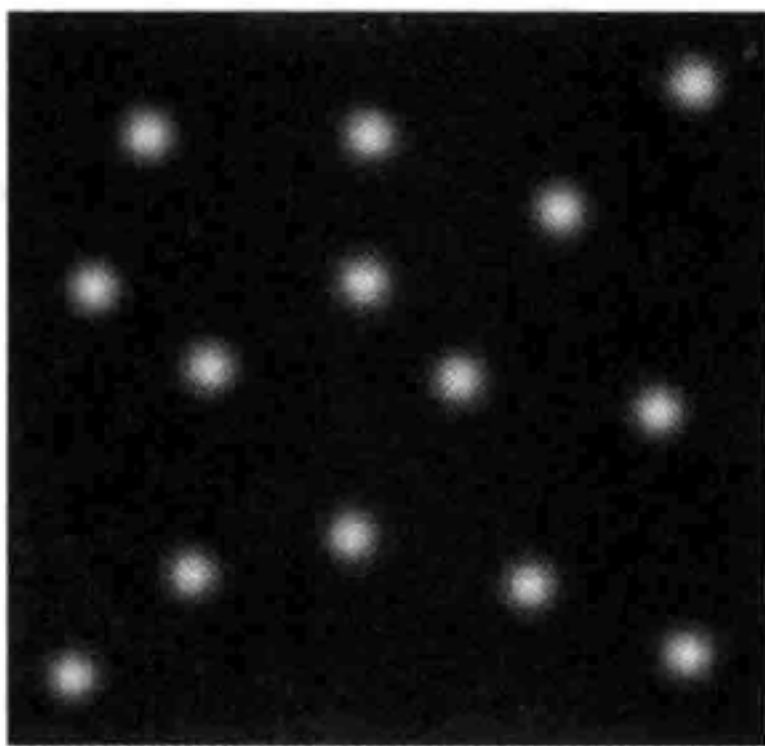


图 11-13 月亮等小星球表面的海拔图

```

package
{
    import away3d.core.utils.Cast;
    import away3d.materials.utils.HeightMapDataChannel;
    import away3d.modifiers.HeightMapModifier;
    import away3d.primitives.Plane;
    import away3d.primitives.Sphere;
    import flash.gemo.Vector3D;
    public class HeightMapModifierDemo extends Away3DTemplate
    {
        [Embed(source="shpere.jpg")]
        protected var SphereTex:Class;
        protected var sphere:Sphere;
        public function HeightMapModifierDemo ()
        {
            super();
        }
        protected override function initScene():void
        {
            super.initScene();
            camera.position=new Vector3D(0,0,500);
            camera.lookAt(new Vector3D(0,0,0));
            sphere=new Sphere(
            {
                segmentsW:32,
                degmentH:32
            }
            );
            scene.addChild(sphere);
        }
    }
}

```

为了将海拔图用于 3D 球,首先要建立一个新的 HeightMapModifier 类的实例,给它提供要修改的 3D 对象、海拔图、读海拔图的彩色通道,以及 3D 对象表面要改变的最大距离。

定义使用彩色通道的类型间有些不同。原来用 Elevation 和 ElevationReader 类定义使用彩色通道时,用的是字符串,HeightMapModifier 类稍有些不同,它使用单位。这里使用表示 HeightMapDataChannel 类的常数值,指定 HeightMapModifier 类从红色通道读。

```

var modifier:HeightMapModifier=new HeightMapModifier(
    sphere,
    Cast.bitmap(SphereTex),
    HeightMapDataChannel.RED,
    16
);

```

然后,execute()函数修改 3D 球的表面高度。

```
modifier.execute();  
}  
}  
}
```

图 11-14 显示了最后的效果。

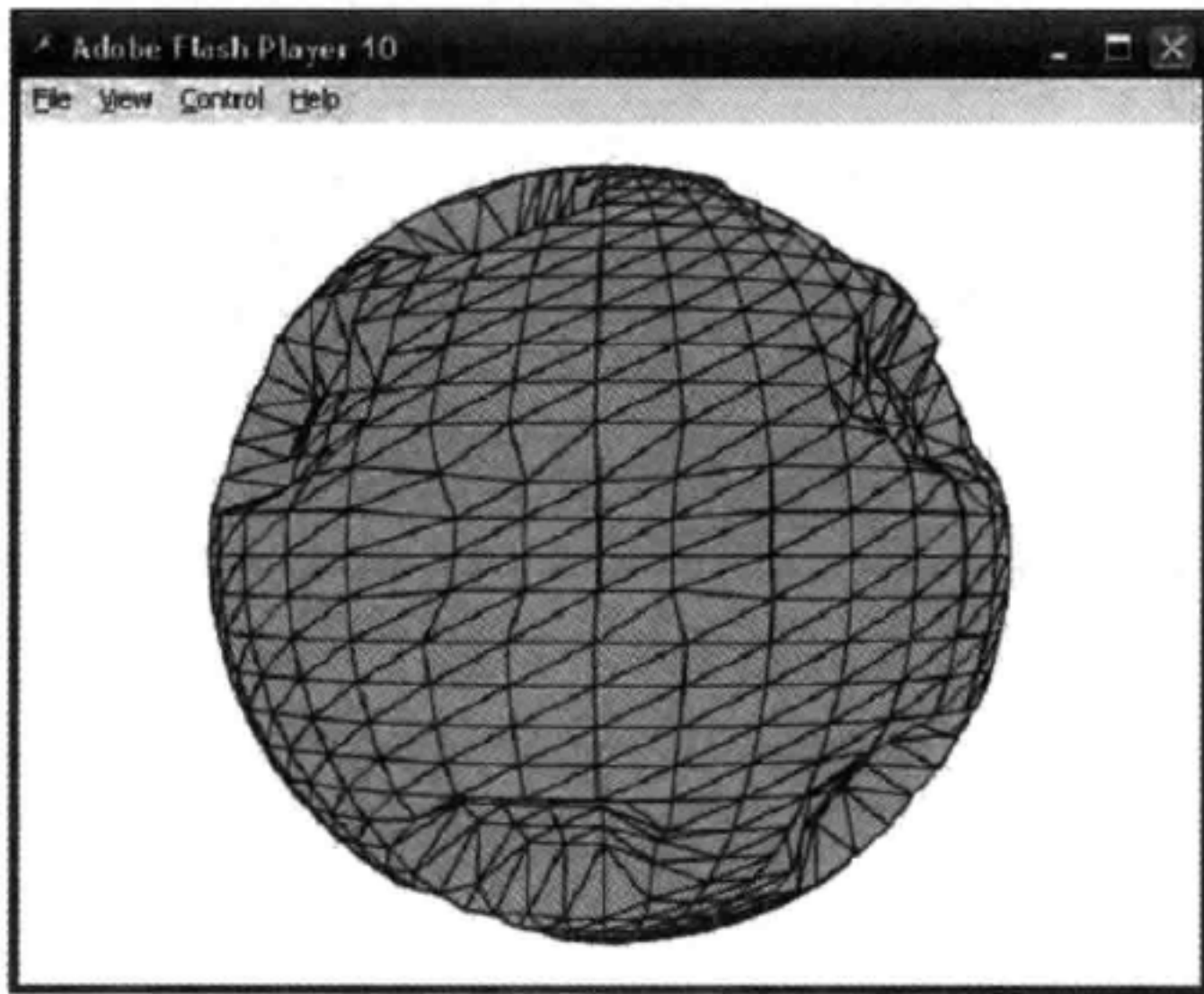


图 11-14 HeightMapModifier 类使用海拔图的效果

读者可能已经注意到,HeightMapModifier 使用海拔图,与 Elevation 类使用海拔图是相反的方式,海拔图上较亮的区,HeightMapModifier 使用在 3D 对象上是凹陷的表面,相反,同样的情况下,Elevation 是增加 3D 对象上表面的高度。解决这个问题的简单方法就是,为 HeightMapModifier 对象构造函数 scale 参数提供一个负值-1,像下面这样:

```
var modifier:HeightMapModifier=new HeightMapModifier(  
    sphere,  
    Cast.bitmap(SphereTex),  
    HeightMapDataChannel.RED,  
    16,  
    -1  
);
```

从海拔图读出的值,改变了反转的标志值(即正值变成了负值),并产生如图 11-15 所示的结果。

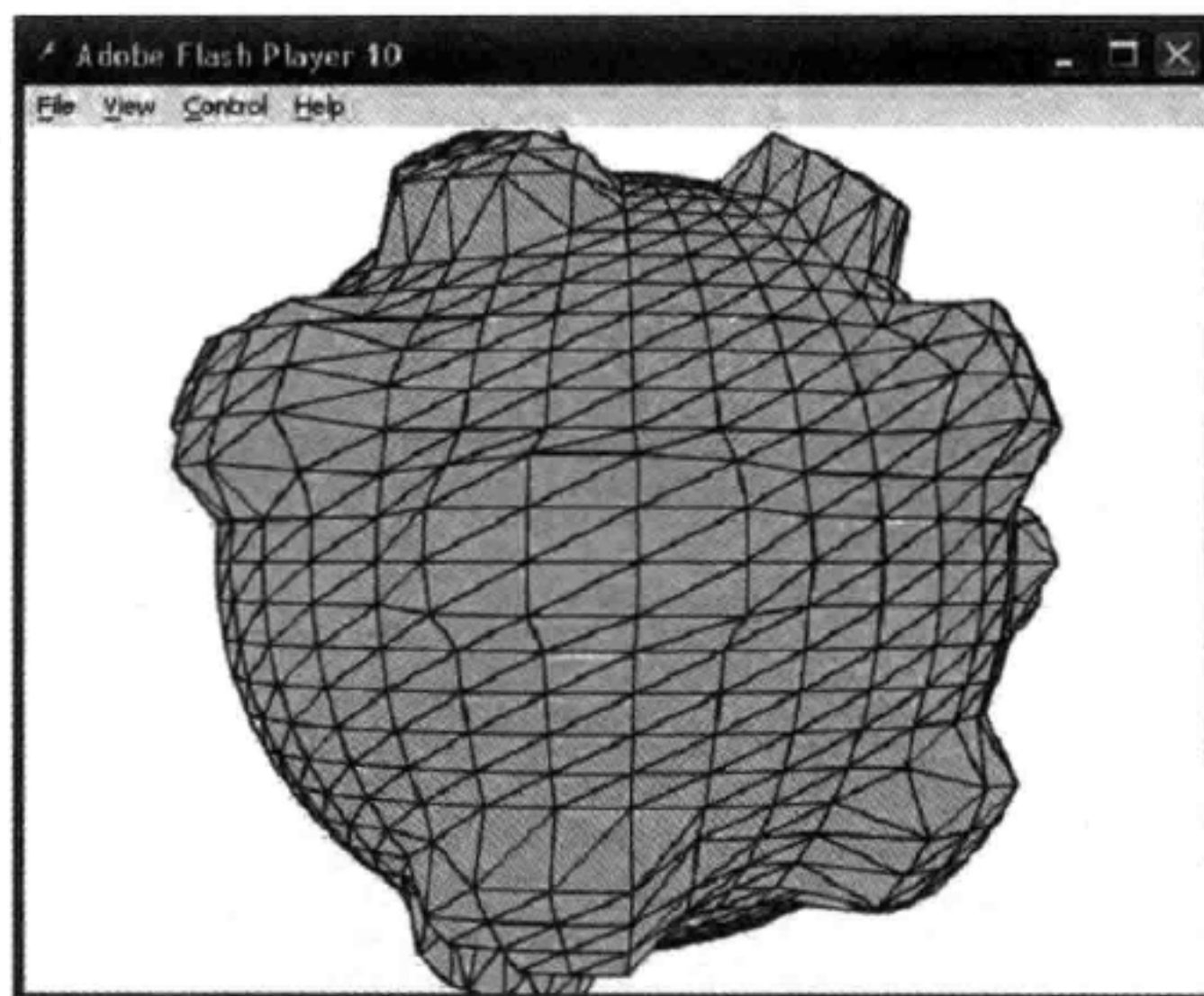


图 11-15 Elevation 类使用海拔图的效果

过滤器和后继处理效果

Away3D 提供了很多显示特效的方法,作为一个 Flash 的开发者,也许已经熟悉 Flash 的 flash.filters 包里的标准过滤器类,Away3D 支持使用这些类,并且它们能应用于各个独立的 3D 对象上或全部的视窗上,显示模糊特效,发出光亮的特效、投影特效。Flash 的最新版本提供了这些新的功能,过滤器类也能应用于 Pixel Bender 着色器和 ShaderFilter 类。

除了由 Flash 提供的过滤器类之外,Away3D 还包含 FogFilter 类,它能应用到场景显示烟雾状的效果,还提供了一些改善运行性能的好处。

最后,将会看到如何使用 BitmapSession 类和 Flash 的各种各样的函数去操作 Bitmap 图像,建立定制的特效。

本章主要内容:

- 把 Flash 过滤器类应用到各个 3D 对象和整个视窗
- 使用 ShaderFilter 类,应用 Pixel Bender 着色器
- 使用 FogFilter 类
- 使用 BitmapSession 类建立一个定制特效的例子

12.1 Flash 和 Away3D 过滤器

Away3D 和 Flash 两者都定义了一些过滤器的类,能用于显示特效,尽管引用的名字相同,但两者之间有重大的区别。

12.1.1 Flash 过滤器

由于 Flash Player 8 的开发者们,在以前的开发中,使用对象的 Filter 属性访问了大量的标准过滤器,而这些过滤器都能用在 DisplayObject 对象上。并且在 Flash Player 10 里引进 Pixel Bender,允许几乎无限制地建立各种过滤器,并应用于使用 ShaderFilter 类。

因为在 Away3D 场景里每个 3D 对象,最终就是通过继承 DisplayObject 类的对象显示在屏幕上的,而标准的 Flash 过滤器又能用在 DisplayObject 类的对象上,这就允许我们在 Away3D 中使用很少的代码,即可创建一些有趣的特效。

1. 过滤器的使用

标准的 Flash 过滤器,能应用到各个独立的 3D 对象上,这时这些 3D 对象的 ownCanvas 属性只要设置为 true 即可。ownCanvas 属性在第 4 章里涉及过,在那里是为强制场景里的 3D 对象景深分类,而把它画在自己的 Canvas 上。正如上面所述,Canvas 是继承 DisplayObject 类的,这就表示,它能让 Flash 过滤器应用到它上面。在下面的程序里,可以有选择地把 Flash 过滤器的特效应用到各个 3D 对象上。

通过把 Flash 过滤器赋值给 View3D 对象一次,将过滤器应用到了所有的可视 3D 对象上。

下面创建一个 OwnCanvasDemo 的应用程序,来证明过滤器能用于各个独立的所有 3D 对象上。

```
package
{
    import away3d.containers.ObjectContainer3D;
    import away3d.core.base.Mesh;
    import away3d.core.utils.Cast;
    import away3d.loaders.Max3DS;
    import away3d.materials.BitmapMaterial;
    import flash.display.BitmapDataChannel;
    import flash.display.Shader;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
```

从 Flash 的 flash.filters 包里输入类,用于显示应用程序要证明的各个特效。

```
import flash.filters.BitmapFilterQuality;
import flash.filters.BlurFilter;
import flash.filters.DisplacementMapFilter;
import flash.filters.DisplacementMapFilterMode;
import flash.filters.GlowFilter;
import flash.filters.ShaderFilter;
```

```
import flash.geom.Point;
import flash.text.TextField;
public class OwnCanvasDemo extends Away3DTemplate
{
```

PBJ 文件要用 Pixel Bender 着色器来编译,在这个例子中使用的 cross-stitch 着色器,是从 Adobe Pixel Bender 交流网站(<http://www.adobe.com/cfusion/exchange/index.cfm?event=productHome&exc=26>)下载的,在这个网站上 Adobe 提供了很多 Pixel Bender 着色器。在这个例子中把嵌入的 crosstitch.pbj 文件作为原始数据文件对待。

```
[Embed(source="crosstitch.pbj", mimeType="application/octet-Stream")]
protected var PixelBenferShader:Class;
```

我们还嵌入了一些其他的文件,包括一个 3DS 模型文件和两个纹理图形文件。

```
[Embed(source="ship.3ds", mimeType="application/octet-Stream")]
    protected var ShipModel:Class;
[Embed(source="ship.jpg")]
protected var ShipTexture:Class;
[Embed(source="displacementmap.jpg")]
protected var DisplacementMap:Class;
```

filterText 属性用于引用 TextField 对象,它用来显示当前过滤器的名字。

```
protected var filterText:TextField;
```

建立两个 3D 对象,一个应用了过滤器,而另一个没有应用过滤器,这就容易将应用了过滤器的与没有应用过滤器的 3D 对象进行比较。

```
protected var shipModel1:ObjectContainer3D;
protected var shipModel2:ObjectContainer3D;
public function OwnCanvasDemo()
{
    super();
}
```

在 initUI() 函数中,对 filterText 属性提供初始化参数。

```
protected override function initUI():void
{
    super.initUI();
    filterText=new TextField();
    filterText.x=10;
    filterText.y=10;
    filterText.width=300;
    this.addChild(filterText);
}
```

```
protected override function initListeners():void
{
    super.initListeners();
    stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
}
```

initScene()函数用于建立并显示两个 3D 对象,并从一个外部模型文件载入 3D 对象。有关这些内容的详述在第 6 章里。

```
protected override function initScene():void
{
    super.initScene();
    this.camera.z=0;
    var shipMaterial:BtmapMaterial=
        new BtmapMaterial(Cast.bitmap(ShipTexture));
    shipModel=Max3DS.parse(Cast.bytearray(ShipModel),
    {
        scale: 0.15,
        z: 500,
        x: 75,
        autoLoadTextures: false,
```

通过设置 ownCanvas 初始对象参数为 true,事实上,就给 3D 对象自己拥有了 Render 进行绘画的 Session。这就允许我们把过滤器用到各个 3D 对象上,而不会影响场景里的其他 3D 对象。

```
ownCavas: true
    }
);
for each (var child:Mesh in shipModel.children)
    child.material=shipMaterial;
scene.addChild(shipModel);
```

作为没有使用过滤器的对比参照,建立第二个 3D 对象,不必设置 ownCanvas 初始化对象参数。

```
shipModel2=Max3DS.parse(Cast.bytearray(ShipModel),
{
    scale: 0.15,
    z: 500,
    x: -75,
    autoLoadTextures: false,
})
);
for each (var child:Mesh in shipModel2.children)
    child.material=shipMaterial;
```



```
scene.addChild(shipModel2);
```

调用 `clearFilters()` 函数,启动不带过滤器的应用程序:

```
clearFilters();  
}
```

当键盘上的键被释放时,调用 `onKeyUp()` 函数,把它用于使用各种过滤器,显示应用程序要证实的特效。

```
protected function onKeyUp(event:KeyboardEvent):void  
{  
    switch(event.keyCode)  
    {  
        case 49:           //1  
            clearFilters();  
            break;  
        case 50:           //2  
            applyBlurFilters();  
            break;  
        case 51:           //3  
            applyDisplacementMapFilters();  
            break;  
        case 52:           //4  
            applyGlowFilters();  
            break;  
        case 53:           //5  
            applyShaderFilters();  
            break;  
        case 54:           //6  
            applyViewBlurFilters();  
            break;  
    }  
}
```

`onEnterFrame()` 函数用于在场景里,围绕 X、Y 轴旋转每个 3D 对象。

```
protected override function onEnterFrame(event:Event):void  
{  
    super.onEnterFrame(event);  
    shipModel.rotationX = shipModel2.rotationX + 2;  
    shipModel.rotationY = shipModel2.rotationY + 1;  
}
```

`clearFilters()` 函数把一个空的数组,赋值给了 `filters` 属性,这个属性是由 `Object3D` 类来定义的,在 `View` 和由 `shipModel` 变量引用的 3D 对象里都有 `filters` 属性。`clearFilters()`

函数有删除过滤器使用的效果。

尽管由 **Away3D Object3D** 类定义的 **filters** 属性,与由 **Flash DisplayObject** 类定义的 **filters** 属性有相同的名字(**Object3D** 类并不是继承 **DisplayObject** 类的),可是,这两个属性能使用在很多相同的场合。当 3D 对象的 **ownCanvas** 属性设置为 **true** 时,应用到 **filters** 属性的数组,最终应用到了 **Render Session** 对象的 **filters** 属性上,用于把 3D 对象画到场景里,制作出在屏幕上的可视的过滤效果。

```
protected function clearFilters():void
{
    filterText.text="None";
    shipModel.filters=[];
    view.session.filters=[];
}
```

2. 模糊过滤器

applyBlurFilter() 函数使用 **BlurFilter** 类,使 3D 对象产生模糊的效果,它能建立表现 3D 对象在焦距外或被雾笼罩的样子。

```
protected function applyBlurFilter():void
{
```

使 **TextField** 对象更新,反映当前使用过滤器的名字。

```
filterText.text="BlurFilter";
```

一个新的 **BlurFilter** 类的实例,先赋值给数组,然后又赋值给 3D 对象的 **filters** 属性。

传递给 **BlurFilter** 对象的结构函数的前两个参数,是 **blurX** 和 **blurY**,定义了沿水平和垂直方向的模糊量。

第三个参数 **quality** 定义模糊效果应用多少次。**BtimapFilterQuality** 类定义了三个常数值: **LOW**, **MEDIUM** 和 **HIGH**。尽管能直接提供最大整数值 15,但较小的值效果渲染得快些。

赋给 **filters** 属性的数组,能包含不只一个 **Flash Filter**,多个 **Flash Filter** 能产生联合的效果。

```
shipModel.filters=[
```

```
new BlurFilter(  
    4,  
    4,  
    BitmapFilterQuality.LOW)  
];  
}
```

图 12-1 显示了 3D 宇宙飞船对象应用模糊过滤器后的效果。



图 12-1 3D 对象应用模糊过滤器的效果

3. 移位映射过滤器

`applyDisplacementMapFilter()` 函数将一个带有新的 `DisplacementMapFilter` 对象的数组赋值给 3D 对象的 `filters` 属性。`DisplacementMapFilter` 类用 `displacement Map` 图像,使 3D 对象表现弯曲,`displacement Map` 图像的彩色值用于设置原图像(或在这种情况下,就是 2D 渲染器中的 3D 对象)中的像素点的偏置定位。这个效果能建立一个带波纹的 3D 对象,如在水下的或纹理玻璃后面的 3D 对象的表现样子。

提供给 `DisplacementMapFilter` 类结构函数的第一个参数是 `mapBitmap`,用于设定从嵌入文件创建的 `BitmapData` 对象。

第二个参数是 `mapPoint`,指定被过滤图像的位置,也就是将要施加置换过滤的图像的左上角,如果想对图像的一部分置换过滤,可重新设置 `mapPoint` 的值,在这种情况下,希望对整个图像进行置换过滤。于是提供一个新的 `Point` 对象,也就是默认的值(0,0)。

第三个和第四个参数是 `componentX` 和 `componentY`,它们给 `displacement Map` 图像提供彩色通道,`displacement Map` 图像既影响图形像素点的 X 位置也影响 Y 位置。但是因为现在的 `displacement Map` 图像是一个灰梯图像,这样红、绿和蓝的彩色通道都是相等的。在这里彩色通道的实际选择对最终的结果不会有什么影响。

第五个和第六个参数是 `scaleX` 和 `scaleY`,定义了一个乘数值,它指定沿 X 坐标和 Y 坐

标有多大的偏置位移量。

第七个参数是 mode, 确定 Flash Player 在由移位像素创建任意的空的空间中应做的事情。DisplacementMapFilterMode 类定义了几个在这里可用的选项。

IGNORE	显示原像素点
WARP(默认值)	围绕另一个图像弯曲像素
CLAMP(在例子中使用)	使用最近的移位像素
COLOR	用指定的色彩填充空间

```
protected function applyDisplacementMapFilter():void
{
    filterText="DisplacementMapFilter";
    shipModel.filters=[
        new DisplacementMapFilter(
            Cast.bitmap(DisplacementMap),
            new Point(),
            BitmapDataChannel.RED,
            BitmapDataChannel.RED,
            15,
            15,
            DisplacementMapFilterMode.CLAMP)
    ];
}
```

图 12-2 显示了应用 DisplacementMapFilter 类的 3D 对象的效果。



图 12-2 3D 对象应用移位映射过滤器的效果

4. 光晕过滤器

applyGlowFilter() 函数应用一个新的 GlowFilter 类的实例, 围绕 3D 对象发出柔和的光辉, 这个效果极大地突出了 3D 对象的选择性, 或建立一个光辉效果。

提供给 GlowFilter 类结构函数的第一个参数是 color, 它定义光辉效果的光颜色。第二个参数是 alpha, 它定义光辉颜色的透明度。第三个和第四个参数是 blurX 和 blurY, 各定义水平和垂直方向上的光辉量。

```
protected function applyGlowFilter():void
{
    filterText="GlowFilter";
    shipModel.filters=[
        new GlowFilter(
            (
                0x4488FF,
                1,
                12,
                12)
            );
    ]
}
```

图 12-3 显示了 GlowFilter 过滤器应用到 3D 对象后的显示效果。



图 12-3 3D 对象应用光晕过滤器的效果

5. Pixel Bender 着色器

Pixel Bender 擅长建立一个像滤镜一样的效果,除了少数包含在 flash.filters 包里的过滤器之外,并且能用来建立大量的各种各样的过滤效果,applyShaderFilter()函数使用了从 Adobe Pixel Bender Exchange 网站下载的十字叉着色器为过滤器。

在第 5 章里已经介绍过 Pixel Bender 在材质上的功能,如 Dot3BitmapMaterialF10、PhongPBMaterial 和 FresnelPBMaterial。Pixel Bender 材质和着色器 Shader 的功能,是依靠 Flash Player 10 的。

```
protected function applyShaderFilter():void  
{  
    filterText.text="ShaderFilter";
```

为了将 Pixel Bender 着色器作为一个过滤器,必须创建一个着色器 Shader 对象,着色器 Shader 的结构函数要有一个代表嵌入的 PBJ 文件类的新实例。

```
var shader:Shader=new Shader(new PixelBenderShader());
```

然后,Shader 对象被传递到一个新的 ShaderFilter 类的结构函数。

```
var shaderFilter:ShaderFilter=new ShaderFilter (shader);
```

ShaderFilter 类允许我们用与其他过滤器类相同的方式,使用 Pixel Bender 着色器,也就是把一个数组赋值给 filters 属性的方式。

```
shipModel.filters=[shaderFilter];  
}
```

图 12-4 显示了十字叉着色器应用到 3D 对象上的效果。



图 12-4 3D 对象应用十字叉着色器的效果

6. 应用过滤器到视窗

过滤器也能应用到整个视窗,而不仅是各个独立的 3D 对象,当过滤器应用到视窗的时候,所有视窗里的 3D 对象都要受到影响,而不管它的 ownCanvas 是否设置为 true。

在这里,把 BlurFilter 赋值给 View3D 对象,使用了将一个数组传递给 filters 属性的相同方法,这最终的效果将是全部 3D 对象表现模糊。

```
protected function applyViewBlurFilter():void  
{  
    clearFilters();
```

```
filterText.text="View BlurFilter";  
view.session.filters=[new BlurFilter()];  
}  
}
```

12.1.2 Away3D 的过滤器

在 Away3D 执行某些任务时候,还包含一些过滤器,这些任务是,对场景里的 3D 对象进行排序,删除那些到照相机的距离比指定值更远的网格元素,或仅保留一定数量的网格元素。此外,FogFilter 类还能在场景里产生雾的效果,对那些离照相机更远的 3D 对象,着以逐渐不透明的彩色。Away3D 的 FogFilter 类要完成这些功能,是将场景细分成小块,然后,对分割的 3D 对象的每个表面着色。图 12-5 阐明了一个带有 7 个这样细分小块的 FogFilter 类。



图 12-5 具有 7 个细分小块的 FogFilter 类实例效果

FogFilter 过滤器还提供了一个运行时的好处,当网格元素处在最新的段的背后时,它们是不渲染的。

下面的 Away3DFilterDemo 类展示了如何将 FogFilter 过滤器应用到场景里,而这个场景包含很多围绕 Y 轴旋转的立方体,旋转运动的各个立方体是由雾状特效的 FogFilter 类创建的。

```
package  
{  
    import away3d.containers.ObjectContainer3D;  
    import away3d.core.filter.FogFilter;  
    import away3d.core.render.BasicRenderer;  
    import away3d.materials.ColorMaterial;
```

```
import away3d.primitives.Cube;
import flash.events.Event;
public class Away3DFilterDemo extends Away3DTemplate
{
    protected var container:ObjectContainer3D;
    public function Away3DFilterDemo()
    {
        super();
    }
    Protected override function initScene():void
    {
        super.initScene();
        this.camera.z=0;
    }
}
```

建立一个 Container3D 类的 container 对象,它用于所有立方体的父容器,这些父容器组成一个场景。

```
container=new ObjectContainer3D(
{
    z: 200
});
scene.addChild(container);
```

场景由很多立方体组成,这些立方体排放在 $6 \times 6 \times 6$ 的格子里,共有 216 个,把每个立方体添加到原先创建的 ObjectContainer3D 容器里,作为它的子部分。

```
for (var primitiveX:int=-50;primitiveX <=50;primitiveX +=20)
{
    for (var primitiveY:int=-50;primitiveY <=50;primitiveY +=20)
    {
        for (var primitiveZ:int=-50;primitiveZ <=50;primitiveZ +=20)
        {
            container.addChild(
                new Cube(
                {
                    x: primitiveX,
                    y: primitiveY,
                    z: primitiveZ,
                    width: 10,
                    height: 10,
                    depth: 10
                }
            )
        )
    }
}
```



```
);  
}  
}  
}
```

在这里,建立一个新的 FogFilter 类的实例。

```
var fogFilter:FogFilter=new :FogFilter(  
{
```

FogFilter 类用 ColorMaterial 类产生雾状的效果,在这里已经建立了一个 ColorMaterial 类的实例,它显示白色,这就表示 ColorMaterial 产生白色雾状的效果。

```
material:new ColorMaterial(0xFFFFFFFF),
```

minZ 参数定义了雾启动时到照相机的距离。

```
minZ: 125,
```

maxZ 参数定义了 3D 对象在还没完全被雾盖住之前(即没有完全被渲染)网格元素到照相机的距离。

```
maxZ: 175,
```

subdivisions 参数定义了有雾效果的有多少个离散层,值越大,显示越平滑,内存消耗也越大。

```
subdivisions: 5
```

```
}  
);
```

尽管在 Away3D 里包含所有可用的渲染器(第 4 章解释了渲染器的更多细节),但这个 filters 属性不是低层渲染器 Renderer 类的一部分,而全部的渲染器类都是低层渲染器 Renderer 类的实现。为此为了要访问 filters 属性,必须对 BasicRenderer 或 QuadrantRenderer 类型组立一个 View3D renderer 的属性。然后就能把 FogFilter 类作为数组的一部分赋值给 filters 属性。

BasicRenderer 总是附加有一个新的 ZSortFilter 对象,它用于场景里 3D 对象的排序,可以把它作为数组赋值给 filters 属性。QuadrantRenderer 类没有附加的作为数组的默认的过滤器 filter,于是当把一个过滤器 filter 赋值给 QuadrantRenderer 时,最好的方法就是每次给它们添加一个 FogFilter,像这样:

```
(view.renderer as QuadrantRenderer).filters.push(FogFilter)
```

```
(view.renderer as BasicRenderer).filter=[fogFilter];  
}
```

围绕 Y 轴旋转容器,也将旋转它里面的并没有雾的子对象,这为 FogFilter 类提供特效的实际证明。

```
protected override function onEnterFrame(event:Event):void  
{  
    super.onEnterFrame(event);  
    ++container.rotationY;  
}  
}
```

在图 12-6 里可看到,着色器是如何应用到网格元素的,这些网格元素是由一些立方体组成的,并且当它们距离照相机距离增大时,立方体数量增加。

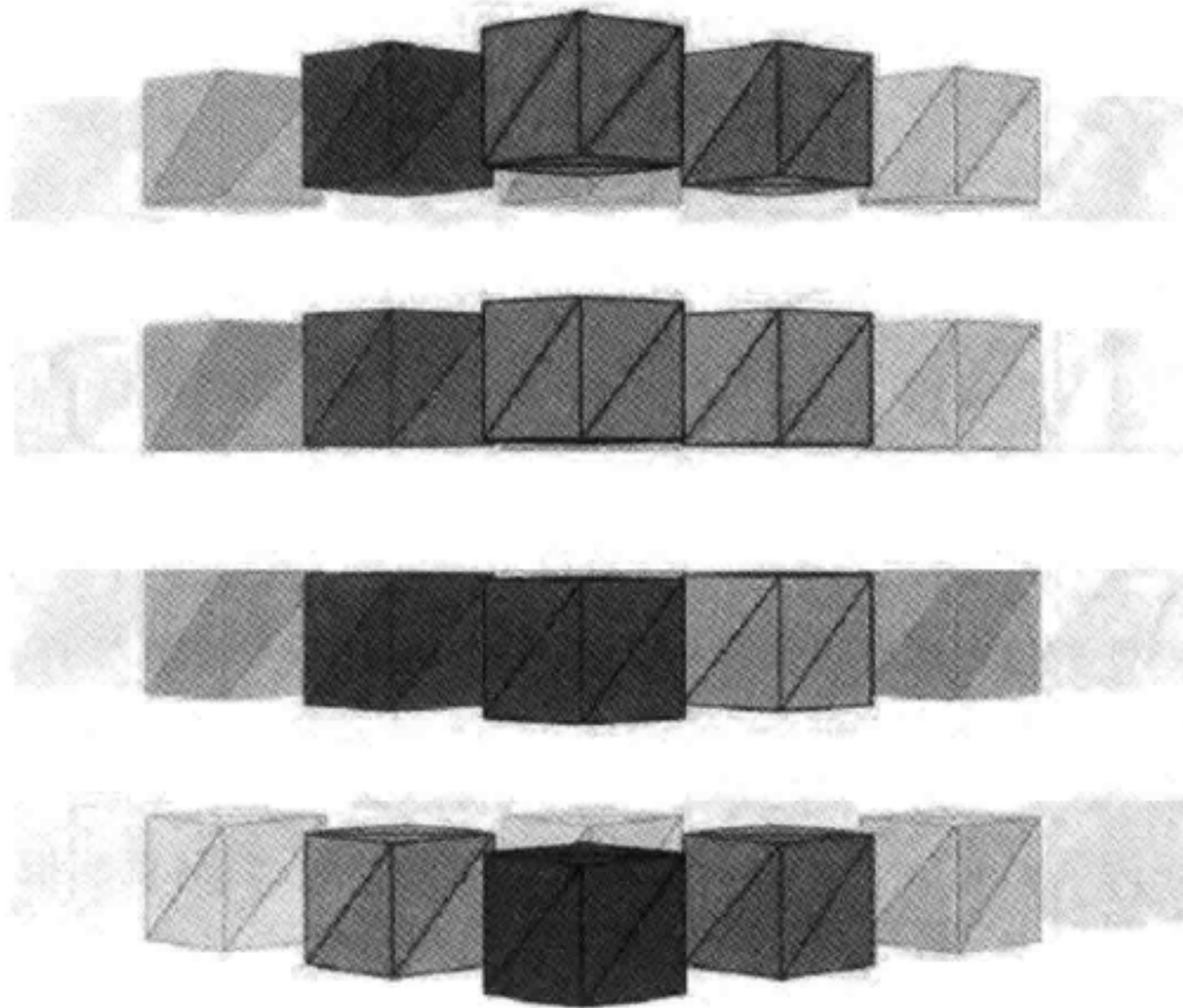


图 12-6 网格元素的着色效果

12.2 渲染器会话对象

默认地,画出每个网格元素,都是用单一的一个渲染会话对象。由视窗和场景使用的默认渲染会话对象是用 SpriteSession 类代表的,因为默认地,每个 3D 对象共享它的父容器渲

染会话对象,也就是说,每个 3D 对象使用的也是场景的 SpriteSession 对象。

Away3D 还包含一个 BitmapSession 类,它提供了一种容易访问位于 Bitmap 下的画出数据,在场景里渲染会话期间,画出这些数据,这些 Bitmap 数据能被访问,也能被修改,这就可以创建一些特别有趣的效果。

为了证实 BitmapSession 类的使用功能,建立一个应用程序,以新的 BitmapSession 对象替换由视窗使用的默认的 SpriteSession 对象。由于我们能用 BitmapSession 类来访问 BitmapData 对象来显示最终的结果,就能使用 BitmapData 对象提供的各种画图程序创建一些特别的效果。

下面的 PostProcessingDemo 应用程序,使用下面的步骤,在场景中表示 3D 飞行器对象起飞时的烟雾效果。

- (1) 将最后一帧转到缓冲区。
- (2) 把缓冲区沿 Y 轴稍为升高一点。
- (3) 修改缓冲区里的像素点颜色,使它们稍为更透明点,并淡入到蓝-灰色。
- (4) 渲染新帧。
- (5) 把新帧画到缓冲区的顶上。
- (6) 把缓冲区画到场景。
- (7) 返回第一步。

```
package
{
    import away3d.containers.ObjectContainer3D;
    import away3d.core.base.Mesh;
    import away3d.core.session.BitmapSession;
    import away3d.core.utils.Cast;
    import away3d.events.ViewEvent;
    import away3d.loaders.Max3DS;
    import away3d.materials.BitmapMaterial;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.filters.BitmapFilterQuality;
    import flash.filters.BlurFilter;
    import flash.geom.ColorTransform;
    import flash.gemo.Matrix;
    import flash.gemo.Point;
    public class PostProcessingdemo extends Away3DTemplate
    {
        [Embed(source="ship.3ds", mimeType="application/octet=stream")]
        protected var ShipModel:Class;
        [Embed(source="ship.jpg")]
        protected var ShipTexture:Class;
        protected var shipModel:ObjectContainer3D;
```

产生的效果依靠图像的最后一帧,图像跨帧转移到下一帧,作为下一帧的背景。buffer 变量引用 BitmapData 对象,而此 BitmapData 对象包含处理过的最后一帧的拷贝。

```
protected var buffer:BitmapData;
```

为了使用 BitmapSession 类,必须建立一个它的实例。BitmapSession 的结构函数使用单一的一个定义大小的参数 scale,令它的值为 1,这就表示 BitmapSession 与视窗 View 有相同的分辨率。

```
protected var bitmapSession:BitmapSession=  
    new BitmapSession(1);
```

为使烟雾效果平滑一些,使用 BlurFilter 类的实例。

```
protected var blur:BlurFilter=new BlurFilter();
```

使用 ColorTransform 类来修改构成图像像素点的彩色和透明度,下面的 ColorTransform 对象将增加彩色的透明度(或减小不透明度),而把 1 加到红色通道,3 加到绿色通道,10 加到蓝色通道。用这种方法修改彩色通道,实际上就产生了一种蓝-灰色的效果。

```
protected var fade:ColorTransform=  
    new ColorTransform(1,1,1,.99,1,3,10,0);
```

下面使用的很多绘画程序,都需要一个 point 对象,最好的方法是在可能的情况下,试着避免创建新对象。因此在这里定义一个新的 Point 类的实例,不给它的结构函数指定参数,其默认值是(0,0)。

```
protected var point:Point=new Point();
```

每一帧的图像都是由上一帧里的像素点转移到屏幕上的。请注意,在这里使用的是 Flash 2D 坐标系统,这就表示把像素点在屏幕上向上移动,就会牵涉它的 2D Y 坐标值的减小。下面的矩阵结构指定了所有参数的默认值,第 6 个除外,它表示沿 Y 轴转换。

```
protected var matrix:Matrix=new Matrix(1,0,0,1,0,-1);  
public function PostProcessingDemo()  
{  
    super();  
}  
protected override function initEngine():void  
{  
    super. initEngine();
```

建立的 BitmapSession 对象将由视窗使用,必须将它赋值给 View3D session 属性。

```
view.session=bitmapSession;
```



```

    }
    protected override function initScene():void
    {
        super. initScene();
        this.camera.z=0;
        var shipMaterial:BitmapMaterial=
            new BitmapMaterial(Cast.bitmap(ShipTexture));
        shipModel=Max3DS.parse(Cast.bytearray(ShipModel),
        {
            autoLoadTextures: false,
            scale: 0.2,
            z: 500
        }
        );
        for each (var child:Mesh in shipModel.children)
            child.material=shipMaterial;
            scene.addChild(shipModel);
    }
    protected override function initListeners():void
    {
        super. initListeners();
    }

```

一旦当前帧完成渲染,就处理产生效果的后几个步骤,也就是第(4)~(7)步。为此注册一个响应中断的函数 `onRenderComplete()`,当产生 `ViewEvent.RENDER_COMPLETE` 中断事件时,便调用它。通过调用 `Away3DTemplate.onEnterFrame()` 函数(调用 `super.onEnterFrame()`)一次,进行后置处理,以实现同样的结果。

```

view.addEventListener(
    ViewEvent.RENDER_COMPLETE,
    onRenderComplete
);
}
override protected function onEnterFrame(event:Event):void
{

```

在当前帧渲染前,处理效果的前几个步骤,也就是第(1)~(3)步。

在通常的情况下的 `onEnterFrame()` 函数里,会立即开始调用 `onEnterFrame()` 函数,在这种情况下,在视窗清屏渲染下一帧之前,我们希望访问最后的那一个渲染过了帧,因此不能立即调用 `super.onEnterFrame(event)`。

首先在场景里旋转 3D 对象:

```

shipModel.rotationX +=2;
shipModel.rotationY +=1;

```

此效果的前几步,也就是第(1)~(3)步,包括最后帧里的 `Bitmap` 数据,用很小的垂直偏

置量和彩色转换把它们画到缓冲里。

我们必须获得最后渲染过的帧的 Bitmap 数据拷贝,这可使用 BitmapSession.getBitmapData()函数,它需要一个当前视窗对象 view 作为其参数。

```
var bitmapSessionData:BitmapData=  
    bitmapSession.getBitmapData(view);
```

如果这是第一次被渲染的帧(即缓冲是空的),就要建立一个与 BitmapSession 的 Bitmap 数据相同大小的缓冲区,这可提供 bitmapSessionData.width 和 bitmapSessionData.height 这前两个参数给 BitmapData 的结构函数。

```
If(buffer==null)  
{  
    buffer=new BitmapData(  
        bitmapSessionData.width,  
        bitmapSessionData.height,  
        true,  
        0  
    );  
}
```

通过画出一个空白的矩形到缓冲区,来清空缓冲区的内容:

```
buffer.fillRect(buffer.rect,0);
```

使用 BitmapData 的 draw()函数,把当前帧的内容复制到缓冲区。

第一个参数 source 是要复制的数据源,要将它设置到从 BitmapSession 取回的 BitmapData 对象,并把它赋值给 bitmapSessionData 变量。

第二个参数矩阵 matrix,用来复制源数据,这里的矩阵把各个像素点变换到屏幕上。

第三个参数 ColorTransform 是一个对象,fade 变量引用了它,ColorTransform 把像素变换成蓝色,并减少像素 alpha 的成分,这使得效果更加透明。

```
buffer.draw(bitmapSessionData,matrx,fade);
```

最后,使用 BlurFilter 过滤器对象,用变量 blur 来引用它。把过滤器用到缓冲区,产生模糊效果。为此使用 BitmapData 的 applyFilter()函数,这个函数要用到以下 4 个参数。

第一个参数是 sourceBitmapData,是要产生特效的 Bitmap 数据源,即所提供的缓冲区 buffer。

第二个参数是 sourceRect,就是输入的 Bitmap 数据源里面的区域,我们希望把这个效果应用到整个图像上,因此需要提供缓冲区的 rect 属性。

第三个参数是 destPoint,它定义了目的图像上的一个点,这个点与源矩形左上角点相对应,它总是 point(0,0),在这个例子里使用之前建立的默认 Point 对象。

第四个参数是 `filter`, 它定义了用来完成过滤器操作的过滤器对象。使用之前建立的用 `blur` 变量引用的 `BlurFilter` 对象。

```
buffer.applyFilter(buffer, buffer.rect, point, blur);
```

现在, 捕获并且修改了最后一帧数据, 继续将它用于渲染下一帧。

```
super.onEnterFrame(event);  
}
```

当前的帧渲染时, 产生特效的后一半过程便完成了。

```
protected function onRenderComplete(event: ViewEvent): void  
{
```

为了继续进行下个过程, 必须得到 `BitmapSession` 对象的下一个 `Bitmap` 数据。

```
var bitmapSessionData: BitmapData =  
    bitmapSession.getBitmapData(view);
```

复制 `Bitmap` 数据, 并把它放到一个新的 `BitmapData` 对象里, 然后把新的 `BitmapData` 对象赋值给 `postProcess` 变量。

```
var postProcess: BitmapData = bitmapSessionData.clone  
();
```

当前帧画到了修改过的最后一帧的顶上。

```
buffer.draw(postProcess);
```

然后, 组成的图像画回到 `BitmapSession`, 这会导致组成的图像画到场景里。

```
bitmapSessionData.copyPixels(buffer, buffer.rect, point);  
}  
}  
}
```

图 12-7 显示了最后的效果。



图 12-7 3D 飞行器起飞时的烟雾效果

Away3D的运行技巧

Flash 平台不断地发展了许多年,它提供了很丰富的利用最新硬件版本的经历。Flash Player 9 引进了 ActionScript 虚拟机 2(ActionScript Virtual Machine 2,AVM2),改善了很多原 Flash Player 版的 AVM1 的运行性能,Flash Player 10 又引进了 Pixel Bender 技术,给开发者提供了更好的运行性能和灵活性。尽管 Away3D 经常透明地利用这些性能,但是仍然有很多技巧能够用来提高 Away3D 应用程序的速度和响应性能。

本章主要内容:

- 确定当前的帧速度
- 设置最大的帧速度
- 设置舞台的质量
- 修改和缩放视口的大小
- 三角缓冲
- 3D 对象的多细节层次
- Away3D 过滤器类
- 3D 模型载入运行

13.1 确定当前的帧速度

当我们谈论有关 Away3D 应用程序运行性能的时候,几乎总要涉及每秒的渲染帧数(Frames Per Second,FPS),也叫帧速率。较高的帧速率将使画面更加流畅,并且更符合最

终用户的视觉体验,尽管能用视觉的办法决定应用程序是否具有可接受的帧速率,但是如果能客观地测量它,那当然是更有用的。幸好,Away3D 具有如此内置的功能。

默认地,Away3D 在结构化的时候,View3D 类就建立了一个 Stats 类的实例,Stats 类在 away3d.core.stats 包里,Stats 对象能通过 View3D 类的 statsPanel 属性来访问,显示 Stats 对象在屏幕上的输出。

为了能看到 Away3D Project stats 选项窗口,必须右击正在屏幕上运行的 3D 对象,这时会弹出一个如图 13-1 所示的 Flash 菜单。

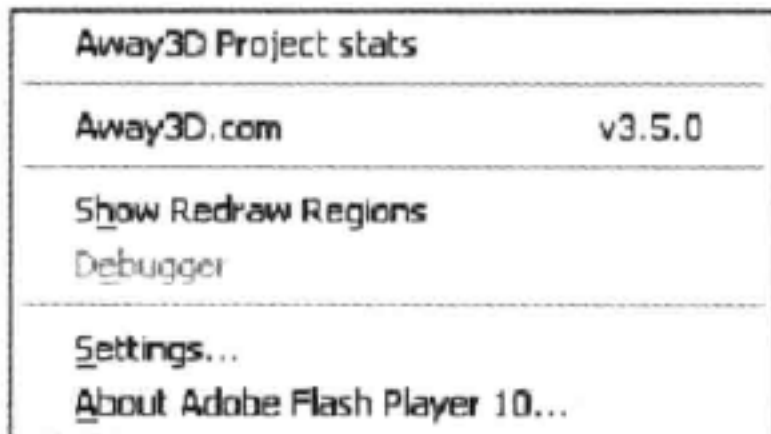


图 13-1 Flash 菜单

再选择 Away3D Project stats 命令,将显示如图 13-2 所示对话框。



图 13-2 Away3D Project 运行对话框

这个对话框中提供了很多测量的数据。

- (1) FPS: 测量当前帧的每秒帧数。
- (2) AFPS: 测量当前帧的每秒平均帧数。
- (3) Max: 测量当前每秒帧的最大峰值。
- (4) MS: 测量最后一个帧渲染用时的毫秒数。
- (5) RAM: 测量应用程序使用了多少内存。
- (6) MESHES: 测量场景里 3D 对象的数量。
- (7) SWF FR: 测量 Flash 应用程序的最大帧速率。
- (8) T ELEMENTS: 测量场景里组成 3D 对象各个元素的总和数。
- (9) R ELEMENTS: 测量场景里被渲染的 3D 对象组成元素的总和数。

当量化一个 Away3D 应用程序运行性能时,这些值将派上用场。

13.2 设置最大的帧速度

Flash 的最新版本,默认的每秒的最大帧数是 24 帧,对于动画这通常是很好的。但是对于游戏,改变到最大的帧速率,可能会觉得更加流畅。使用 SWF 的 `frameRate` 元标签完成这个工作是非常方便的,把以下一行代码添加到第 1 章介绍过的 `Away3DTemplate` 类的前面即可。

```
[SWF(frameRate=100)]
public class Away3DTemplate extends Sprite
{
    //class definition goes here
}
```

SWF FR 测量并显示 **Away3D Stats** 对象,反映 `frameRate` 元标签定义的最大帧速度。

请注意,使用 `frameRate` 元标签定义的最大帧速度,并不总是表示应用程序运行在最高的帧速度,仅表示它可在最高的帧速度运行。一台运行速度慢的个人计算机,即使已设置最大的帧速度为最高帧速,但 `Away3D` 的应用程序仍将在慢的帧速度下运行。

必须要注意的是,在 `onEnterFrame()` 函数里,如 3D 对象的转换,运行转换计算是从属于应用程序的帧速度的。在下面的代码中,旋转 3D 对象围绕 X 轴每帧 1° ,像这种转换,在本书提供的例子里到处都是。

```
override protected function onEnterFrame(event:Event):void
{
    super.onEnterFrame(event);
    shipModel.rotationX += 1;
}
```

如果帧速是 30 FPS,3D 对象将围绕 X 轴每秒转 30° ,如果帧速是 90 FPS,3D 对象将围绕 X 轴每秒转 90° 。如果应用程序坚持需要上面代码里的这样一种旋转转换,而不管帧速,可使用 `Tweening` 库,像第 3 章里展示的那些例子。

13.3 设置 Flash 舞台的质量低一点

读者可能已经注意到了,Flash 在它的右键菜单里提供了很多质量设置选项,可从 4 个选项里选择一个,`StageQuality` 类位于 `flash.display` 包里,正如 Flash API 文档描述的那样,

这些设置的作用如下。

(1) StageQuality.LOW 指定低渲染呈现品质：不消除图形的锯齿，不对位图进行平滑处理。但是运行时仍在使用多重贴图。

(2) StageQuality.MEDIUM 指定中等呈现品质：使用 2×2 像素网格消除图形锯齿，但对位图进行平滑处理依赖 Bitmap.smoothing 的设置，此设置适用于不包含文本的影片，但是运行时仍在使用多重贴图。

(3) StageQuality.HIGH 指定高呈现品质：使用 4×4 像素网格消除图形锯齿，对位图进行平滑处理依赖 Bitmap.smoothing 的设置，并且如果影片是静态的，则还对位图进行平滑处理。运行时仍在使用多重贴图。这是 Flash Player 使用者默认的渲染质量设置。

(4) StageQuality.BEST 指定极高呈现品质：使用 4×4 像素网格消除图形锯齿，并且始终对位图进行平滑处理。

Mip-mapping 使用了多重映射，它预先计算出原 Bitmap 图的较小版本，当原图缩小到 50% 以下时，使用 Bitmap 图的较小版本而不使用 Bitmap 原图，当带有 Bitmap 材质的 3D 对象在场景里按距离比例缩小时这种情况就可能发生。

质量设置是把上述值中的一个赋值给舞台对象的 quality 属性：

```
stage.quality=StageQuality.LOW;
```

Away3D 设置舞台质量提供的一些样本例子是使用 SWF 的 quality 元标签：

```
[SWF(quality="LOW")]
```

Flex 编译器不支持使用这种方法设置舞台质量，尽管代码在编译时不会报错，舞台质量仍然维持默认的 StageQuality.HIGH 值。可在以下网址找到有关 Flex 编译器使用元标签的更多资料 http://livedocs.adobe.com/flex/3/html/help.html?content=metadata_3.html。

设置舞台质量低些，将会改善应用程序运行时的性能，增加了在应用程序中显示大量 3D 对象的感受，如第 9 章里 DOFSpriteDemo 应用程序那样。

设置舞台质量低呈现下降趋势，是因为它影响舞台上的所有 3D 对象，不仅是 Away3D 画出的那些。当画出文本时，设置舞台质量低要特别注意，如文本域和按钮的可视控制会降低。

使用中等的舞台质量设置,是在运行速度与可视质量之间折中的好办法。

13.4 减小视口尺寸的大小

画在屏幕上的像素点越少,渲染过程就越快,画进视窗的区域,能通过把一个 RectangleClipping 对象赋值给 View3D 类的 Clipping 属性来定义。

为了能使用 RectangleClipping 类,首先必须从 away3d.core.clip 包里输入它,然后,定义 Away3D 把 RectangleClipping 类画到视窗的区域,这要给 RectangleClipping 的结构函数提供初始化参数 minX,maxX,minY 和 maxY 的值来完成。

```
view.clipping=new RectangleClipping(  
  {  
    minx: -100,  
    maxX:100,  
    minY: -100,  
    maxY: 100  
  }  
);
```

上面的代码将限制视窗的输出区域为 200×200 。

ViewportClippingDemo 应用程序,允许在程序运行时,用箭头向上键和向下键修改剪接板矩形的大小。在图 13-3 中,可见到剪接板矩形制作出的不同效果,左边的剪接板矩形设置为舞台的满屏,右边的剪接板矩形设置减小了。

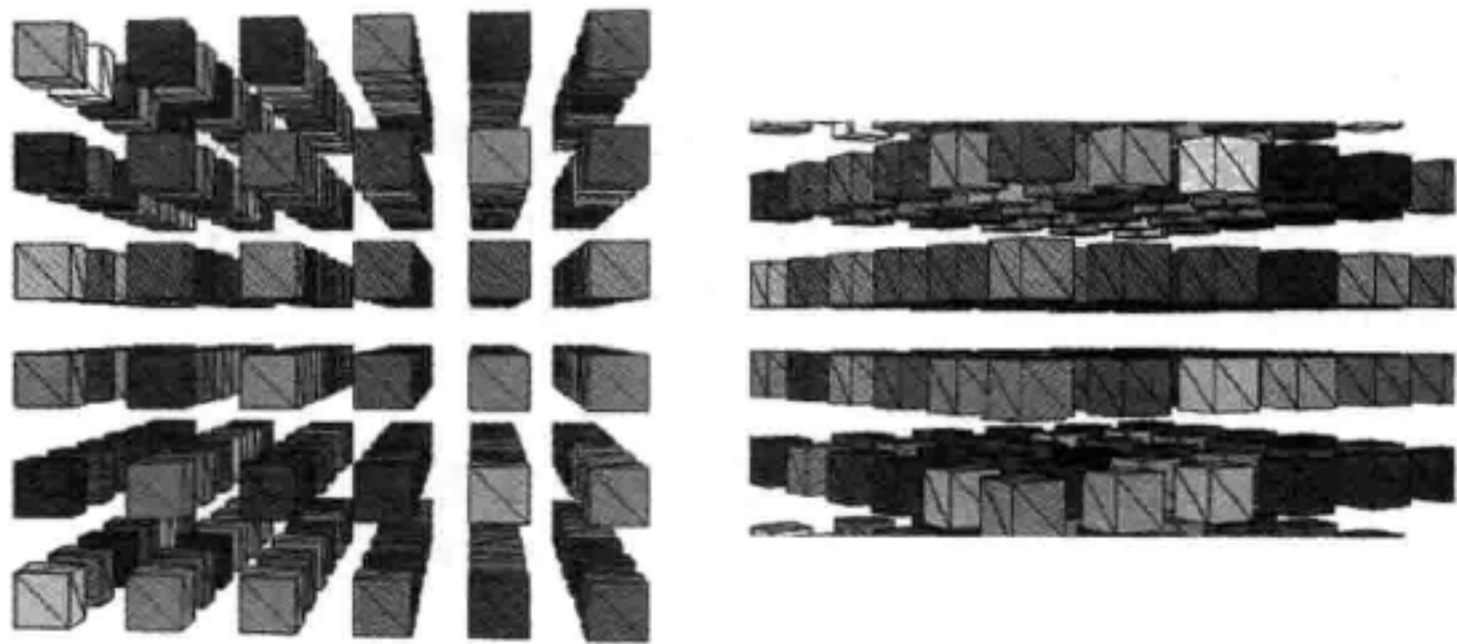


图 13-3 剪接板矩形的不同效果

13.5 缩放视口输出

减少 Away3D 渲染像素点数量的另一种方式,是把 BitmapSession 对象赋值给 View3D 类的 session 属性,传递到 BitmapSession 结构函数的 Number,定义了画到场景里 Bitmap 的内置缩放比例。默认的内置缩放比例是 2,这将要建立一个宽度和高度是视口宽度和高度一半的内部 Bitmap,这就表示最后的内部 Bitmap 尺寸是视口的 1/4,因此仅 1/4 的像素需要画出。

为了能使用 BitmapSession 类,首先必须从 away3d.core.session 包里输入它,然后把一个新的 BitmapSession 对象赋值给 View3D 类的 session 属性。

```
view.session=new BitmapSession(2);
```

当图像画出时,内部 Bitmap 将放大到全舞台,不像 Clipping 视口,因此使用 BitmapSession 类允许应用程序显示整个可用的区域,这将导致像素化的图形。在图 13-4 中能看到像素化的结果。使用 BitmapSession 类画出左边的图形,scale 是 1。而使用 BitmapSession 类画出右边的图形,scale 是 2。

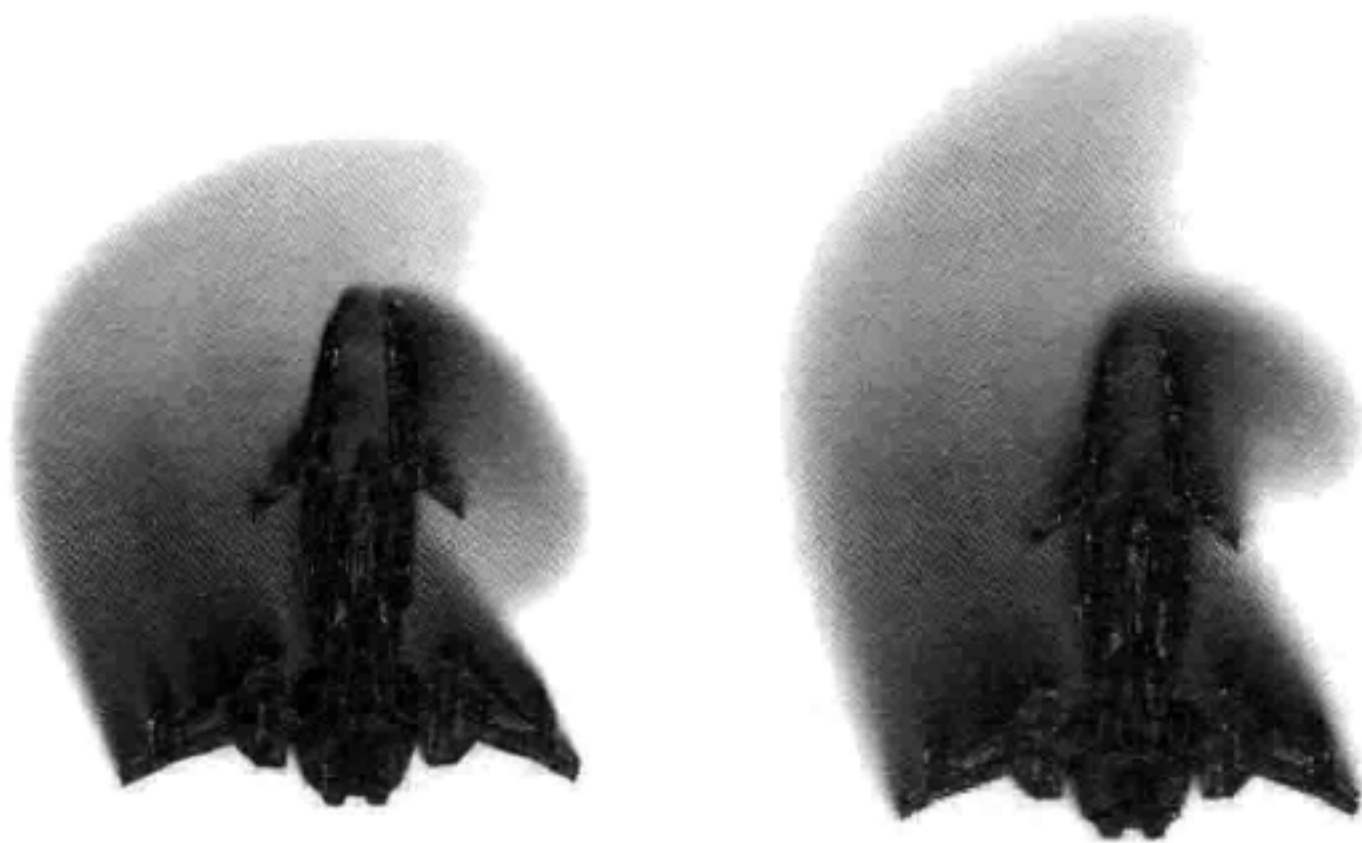


图 13-4 3D 对象像素化的效果

13.6 三角缓存

Away3D 有一个三角缓存功能,默认情况下即启用。在最后一帧,如果画了 3D 对象的 canvas 还没有到修改的时候,三角缓存使 3D 对象不再画出来。

默认地,所有的 3D 对象都画到了一个共用的 canvas,在这样的情况下,如果场景里没有要修改的 3D 对象,三角缓存系统提供了一些好处,这就是将导致巨大的速度提升静态的 3D 场景,以很高的帧速画出静态的 3D 场景。实际上这对我们帮助并不大,因为仅能容易显示一些静态的图像而已。而三角缓存可以单独启用各个 3D 对象,只要把它的 ownCanvas 属性设置为 true 即可,或者,同样的道理,三角缓存也可以单独启用各个 3D 对象组,只要把 3D 对象组添加到 ObjectContainer3D 对象的子容器,并使 ownCanvas 属性设置为 true 即可。在这样的工作模式下,每帧里的重新画出的 3D 对象就界定到各个被修改过的 3D 对象或者那些被修改过的子容器。上述的这些情况的出现,就是因为修改了照相机的位置,引起每个 3D 对象要重新画出。

当照相机是静态的时候,或者给定帧里只有少量的 3D 对象需要修改的时候,三角缓存是非常有用的。TriangleCachingDemo 应用程序通过把 75 个复杂的 3D 对象添加到场景,展示了三角缓存的功能。每个复杂的 3D 对象都有它自己的 ownCanvas 属性,并设置为 true,使三角缓存能应用到各个复杂的 3D 对象,每个复杂的 3D 对象都能响应鼠标事件,当鼠标光标移动到复杂的 3D 对象下时,能响应 MouseEvent3D. MOUSE_OVER 或 MouseEvent3D. MOUSE_OUT 事件而画出它们。在任意时候修改大量的 3D 对象,因此引起每帧的大量的 3D 对象要重新画出。三角缓存允许这个应用程序以较高的帧速运行,比每帧里重新画出每个 3D 对象的速度要快。

```
package
{
    import away3d.containers.ObjectContainer3D;
    import away3d.core.base.Mesh;
    import away3d.core.utils.Cast;
    import away3d.events.MouseEvent3D;
    import away3d.loaders.Max3DS;
    import away3d.materials.BitmapMaterial;
    import flash.events.Event;
    import flash.filters.GlowFilter;
    import flash.utils.getTimer;
    public class TriangleCachingDemo extends Away3DTemplate
    {
        [Embed(source="ship.3ds", mimeType="application/octet-stream")]
        protected var ShipModel:Class;
        [Embed(source="ship.jpg")]
        protected var ShipTexture:Class;
        protected static const MESH_SCALE:Number=0.02;
        protected static const SCAL_FACTOR:Number=Math.PI/1000;
        protected var selectedMesh:ObjectContainer3D;
        public function TriangleCachingDemo()
        {
            super();
```

```
}  
protected override function initScene():void  
{  
    super.initScene();  
    this.camera.z=0;
```

从嵌入式文件载入初始化 3D 模型和纹理,我们将把这些用为添加到场景里的所有 3D 对象的模板。

```
var shipMaterial:BitmapMaterial=  
    new BitmapMaterial(Cast.bitmap(ShipTexture));  
var shipModel:ObjectContainer3D=  
    Max3D.parse(Cast.bytearray(ShipModel),  
    {  
        autoLoadTextures: false,  
        scale: MESH_SCALE,  
        rotationY: 180,
```

为了使三角缓冲系统能工作,必须设置这些模型的 ownCanvas 属性为 true。

```
        ownCanvas=true  
    }  
);  
for each (var child:Mesh in shipModel.children)  
    child.material=shipMaterial;
```

在这里,使用 for 的三重循环,每次计算出添加到场景里各个 3D 对象的 x、y 和 z 的坐标值。

```
var meshClone:ObjectContainer3D  
for (var xPos:int=-40; xPos<=40; xPos+=20)  
{  
    for (var yPos:int=-40; yPos<=40; yPos+=20)  
    {  
        for (var zPos:int=100; zPos<=180; zPos+=40)  
        {
```

然后克隆 3D 对象的模板,这为我们节省了解析 3D 对象模型文件的时间(稍后会看到这是相当有效的),也节省了一些内存,因为每个被克隆的 3D 对象,引用的是 3D 对象的几何形态的模板,而不是维持它们自己的数据拷贝。

```
meshClone=shipModel.clone() as ObjectContainer3D;
```

然后,设置被克隆的 3D 对象的位置,并添加到场景。

```
meshClone.x=xPos;  
meshClone.y=yPos;
```

```
meshClone.z=zPos;
scene.addChild(meshClone);
```

每个被克隆的 3D 对象都要设置为能响应鼠标 MouseEvent3D.MOUSE_OVER 和 MouseEvent3D.MOUSE_OUT 事件。

```
meshClone.addEventListener(
    MouseEvent3D.MOUSE_OVER,
    onMouseOver
);
meshClone.addEventListener(
    MouseEvent3D.MOUSE_OUT,
    onMouseOut
);
}
}
}
protected function onMouseOver(event:Event3D):void
{
```

当鼠标光标放到一个 3D 对象上时,当前的 3D 对象被选中,设置被选中 3D 对象的属性是 selectedMesh。

```
selectedMesh=event.target as ObjectContainer3D;
```

GlowFilter 类用于形象地指定被选的那个 3D 对象。

```
selectedMesh.filters=[new GlowFilter()];
}
```

当鼠标光标移出一个 3D 对象上时,表示没有当前的 3D 对象被选中,设置 selectedMesh 为 null 反映这一事实。

```
protected function onMouseOut(event:Event3D):void
{
    selectedMesh=null;
    var mesh:ObjectContainer3D=
        event.target as ObjectContainer3D;
```

3D 对象的 filters 属性设置为空数组,这样来清除在 onMouseOver() 函数中已经赋值过了的 GlowFilter 对象。

```
mesh.filter=[];
```

将 3D 对象的比例设置回其默认值。


```

mesh.scale(MESH_SCALE);
}
protected override function onEnterFrame(event:Event):void
{
    super.onEnterFrame(event);
    if (selectedMesh != null)
    {

```

如果有一个 3D 对象在鼠标指针下,使用一些简单的数学计算合并 3D 对象的大小在 1 与 2 之间,这展示各个 3D 对象能够被变换或者被活动起来。而其他的余下的静态 3D 对象,因为使用了三角缓存而保持高的帧速度。

```

var betweenNegOneAndOne:Number=
    Math.sin(getTimer()*SCALE_FACTOR);
var betweenZeroAndOne:Number=
    (betweenNegOneAndOne+1)/2;
var betweenOneAndTwo:Number=
    betweenZeroAndOne+1;
selectedMesh.scale(betweenOneAndTwo*MESH_SCALE);
    }
}
}
}

```

13.7 模型的细节层次

模型的细节层次是一种技术,当模型离背景远时,用少量的多边形数量显示较简单的模型。当模型靠近照相机的时候,用较多的多边形数量,显示质量高些的模型。在背景里,那些牺牲了显示质量的 3D 对象,运行性能有明显提高。并且,因为那些远些的 3D 对象画到屏幕上时小一些,因而,时常不会感到有明显的视觉质量下降。

away3d.containers 包里的 LODObject 类是一种容器,在其子容器中的 3D 对象处在一定的透视值范围时,LODObject 类能显示它们。

把若干的 LODObject 对象编成一组,用于实现模型的细节层次技术。作为一个 LODmodelsDemo 例子,我们使用三个原始球,每个结构使用不同数量的多边形。

```

var sphere0:Sphere=new Sphere(
{
    radius:50,
    segmentsW: 4,
    segmentsH: 3
}

```

```
);  
var sphere1:Sphere=new Sphere(  
{  
    radius:50,  
    segmentsW: 10,  
    segmentsH: 8  
})  
);  
var sphere2:Sphere=new Sphere(  
{  
    radius:50,  
    segmentsW: 16,  
    segmentsH: 12  
})  
);
```

然后分别把这三个球添加成新的 LODObject 对象的子对象,minp 初始化参数定义了 LODObject 对象的子对象在可视范围内的最小值,通过提供的 maxp 参数的值,定义了可视范围内的最大值(但不包含最大值)。

```
var lodObject0:LODObject=new LODObject(  
{  
    minp: 0,  
    map: 0.25  
},  
sphere0  
);  
var lodObject1:LODObject=new LODObject(  
{  
    minp: 0.25,  
    map: 0.5  
},  
sphere1  
);  
var lodObject2:LODObject=new LODObject(  
{  
    minp: 0.5,  
    map: 1  
},  
sphere2  
);
```

使这三个 LODObject 对象作为一个组,把它们添加到一个标准的 ObjectContainer3D 对象的子对象。

```
var container:ObjectContainer3D=
```

```
new ObjectContainer3D(lodObject0, lodObject1, lodObject2);
```

再把容器添加到场景里,当 ObjectContainer3D 对象(和它的子 LODObject 对象)和照相机间的距离改变时,这三个球即可看到。

使用照相机默认的属性值,放大率 10,焦距 100,能计算出 3D 对象 sphere0 在可视时,离相机的距离大于 3900 单元。3D 对象 sphere1 在可视时,离相机的距离小于 3900 单元大于 1900 单元。3D 对象 sphere2 在可视时,离相机的距离小于 1900 单元大于 0 单元。

当程序运行的时候,图 13-5 中显示了含有 LODObject 对象的容器到照相机的距离。

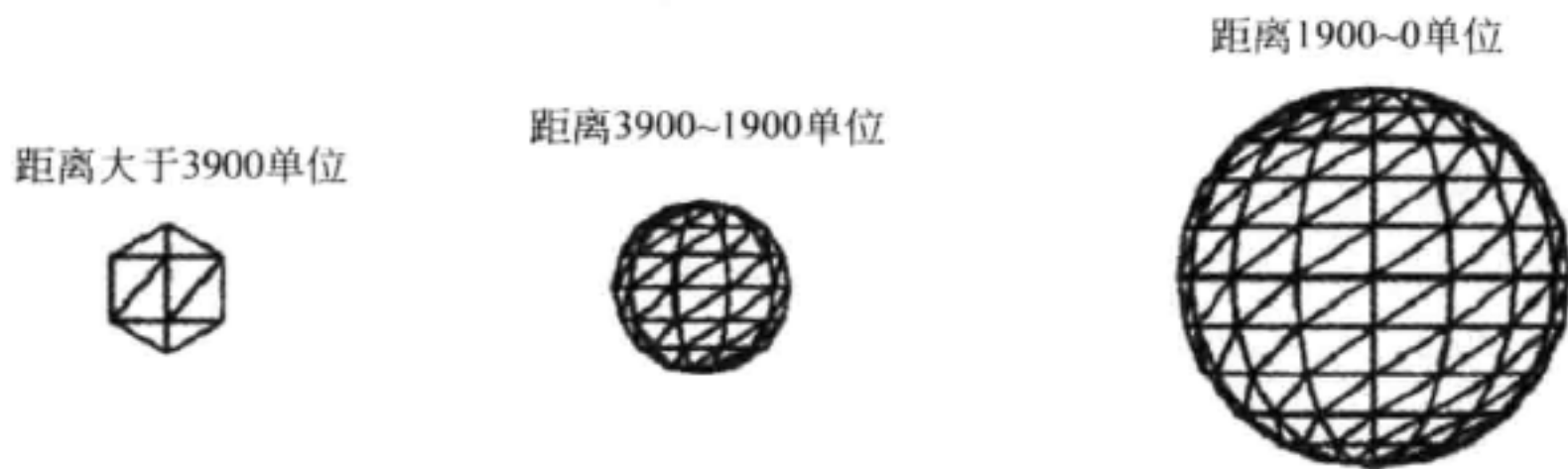


图 13-5 3D 对象与照相机的距离显示

13.8 Away3D 过滤器

在第 12 章里,已看到 FogFilter 类是如何添加到场景里产生模糊效果的。也应注意到,在模糊最后一层以后的那些 3D 对象,它们根本就不画出,如此可以提升运行性能。

在 away3d. core. filters 包里,还有两个附加的过滤器类:MaxPolyFilter 和 ZDepthFilter,它们都不添加可视的效果,而都能用于减少画到场景里的网格数量。

这两个类都可用于如第 2 章中所介绍的 FogFilter 类里,把它们赋值给 BasicRenderer 或 QuadrantRenderer 类的 filters 属性。或者,把它们传递给 BasicRenderer 或 QuadrantRenderer 类的构造函数,然后,这两个渲染类赋值给 View3D 类的 renderer 属性。

13.8.1 Z 景深过滤器

Z 景深过滤器定义了网格的最大景深,凡是那些大于最大景深的网格,就不会画到场景里,这给 FogFilter 类提供了一些运行时的好处,但不会产生模糊效果。

ZDepthFilter 的构造函数有一个参数 maxZ,它定义了要被剔除的那些网格元素到照相机的距离。在下面的例子里,建立了一个新的 ZDepthFilter 类的实例,它将剔除那些到照相机的距离比 200 单元大的网格元素。

```
var zDepthFilter:ZDepthFilter= new ZDepthFilter(200);  
view.renderer=new BasicRenderer(zDepthFilter);
```

Away3D 3.6 版本的 ZDepthFilter 有一个 bug, 为避免 ZDepthFilter 的 bug, 改用 away3d.core.clip 包里的 FrustumClipping 类, 它能完成同样的效果, 但不会给运行带来好处。

```
view.clipping=new FrustumClipping({maxZ:200});
```

13.8.2 最大多边形过滤器

MaxPolyFilter 过滤器, 仅给予一个设置画到场景里的网格数量, 它做这样的工作, 是从将要画到场景里的元素集里, 记住一个许可画到场景中的网格数量。因为将要画到场景里的元素集是按 Z 景深排序的, 所以 MaxPolyFilter 过滤器有这样的效果: 它将丢弃那些场景里表示最远的 3D 对象的网格元素。

MaxPolyFilter 的构造函数有一个参数 maxp, 它定义了有多少网格元素可画到场景里。下面的代码创立了一个新的 MaxPolyFilter 类的实例, 它仅许可画出到场景最近的 500 个网格元素。

```
var maxPolyFilter:MaxPolyFilter=new MaxPolyFilter(500);  
view.renderer=new BasicRenderer(maxPolyFilter);
```

赋给 maxp 属性的值, 直接与状态面板显示的 R ELEMENTS 值相对应。

13.8.3 在后台进行画图

在 3D 应用程序里, 有很多的例子要画出大量的相似的 3D 对象, 如一群鱼、一群人或城市里很多的大厦, 这可通过多次画出各个 3D 对象来建立。

后台进行画图能非常快速地创建这种类型的场景。考虑图 13-6。

即使这个场景是由几百个建筑物构成, 但每个建筑物用了 5 种不同的模型的其中一种来显示, 但因为每个建筑物是座落在一个单一的平面上, 这意味着每个建筑物都能用大概相同的视角看到, 后台进行画图便能用在这样的情形, 使性能得以大幅度的提升。

后台进行画图的概念是: 由视口画出的 3D 对象, 还没有加到舞台, 因此是看不到的(或“后台画”)。视口画出 3D 对象图像, 然后, 由很多 Sprite3D 对象, 在场景里可视地显示出来。渲染一个 3D 对象并在多个 Sprite3D 对象上显示出来, 其速度要比多次画出原各个 3D 对象快得多。

为了证实在 Away3D 里, 后台进行画图是如何实现的, 下面建立一个叫做

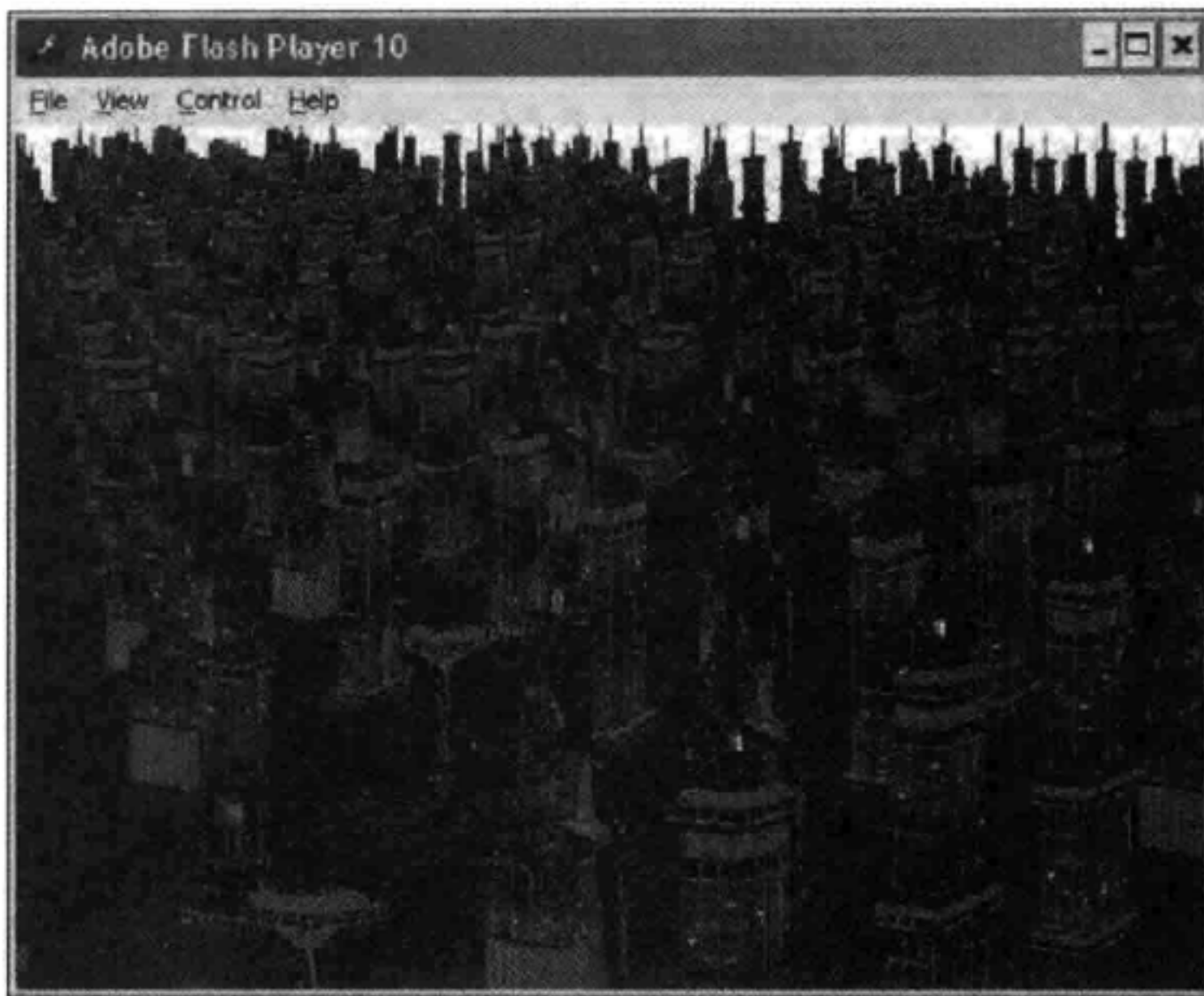


图 13-6 城市场景

OffscreenRenderingDemo 的应用程序。

```
package
{
    import away3d.cameras HoverCamera3D;
    import away3d.containers.ObjectContainer3D;
    import away3d.containers.View3D;
    import away3d.core.base.Mesh;
    import away3d.core.clip.RectangleClipping;
    import away3d.core.session.BitmapSession;
    import away3d.core.utils.Cast;
    import away3d.loaders.Max3DS;
    import away3d.materials.BitmapMaterial;
    import away3d.primitives.Plane;
    import away3d.sprites.Sprite3D;
    import flash.display.BitmapData;
    import flash.display.StageQuality;
    import flash.events.Event;
    import flash.events.MouseEvent;
    [SWF(background="FFFFFF")]
    public class OffscreenRenderingDemo extends Away3DTemplate
    {
```

从嵌入文件输入用于建筑物和地平面的纹理图。

```
[Embed(source="building.jpg")]
protected var BuildingTexture:Class;
[Embed(source="ground.jpg")]
protected var GroundTexture:Class;
```

同样,下面的这些 3ds 文件是建筑物的模型。

```
[Embed(source="building1.3ds", mimeType="application/octet-stream")]
protected var Building1:Class;
[Embed(source="building2.3ds", mimeType="application/octet-stream")]
protected var Building2:Class;
[Embed(source="building3.3ds", mimeType="application/octet-stream")]
protected var Building3:Class;
[Embed(source="building4.3ds", mimeType="application/octet-stream")]
protected var Building4:Class;
[Embed(source="building5.3ds", mimeType="application/octet-stream")]
protected var Building5:Class;
```

每个建筑模型,都要添加到它自己的视口。buildingViews 数组集合了对这些各后台画视口的引用。

```
protected var buildingViews:Vector.<View3D>=
    new Vector.<View3D>();
```

这 5 个视口的每一个都将用来建立各自的 BitmapMaterial 对象,这些 Bitmap 图像对象以后将用来显示在各自的 Sprit3D 对象上。billboardMaterial 数组集合了对这些 Bitmap 图像材质的引用。

```
protected var billboardMaterial:Vector.<BitmapMaterial>=
    new Vector.<BitmapMaterial>();
```

mouseButtomDown、lastStageX 和 lastStageY 属性,用于使旋转相机转动。

```
protected var mouseButtomDown:Boolean=false;
protected var lastStageX:Number=0;
protected var lastStageY:Number=0;
public function OffscreenRenderingDemo()
{
    super();
}
```

initEngine()函数用来设置舞台质量为低,建立旋转照相机和通过调用 buildOffscreenViews()函数建立一些后台画的视口。

```
protected override function initEngine():void
```

```
{
    super. initEngine();
    stage.quality=StageQuality.LOW;
    view.camera=new HoverCamera3D(
    {
        distance: 1500,
        yfactor: 1,
        tiltAngle: 15
    }
    );
    for (var i:int=0; i<5; ++i)
        buildingViews.push(builOffscreenView());
}
```

通过 buildOffscreenViews() 函数, 后台画的视口被建立起来。

```
protected function buildOffscreenView():View3D
{
```

建立的各个后台画的视口, 不同于建立正规的视口, 都以 View3D 类表示。

```
var buildingView:View3D=new View3D();
```

尽管并不实际地把后台画视口加到舞台, 但仍然必须要给视口单位, 犹如它在视口可视的一样。

```
buildingView.x=stage.stageWidth/2;
buildingView.y=stage.stageHeight/2;
```

为了取到视口的输出, 并显示它为材质, 必须使用 BitmapSession 类。BitmapSession 类的渲染器把 BitmapMaterial 类渲染为源 Bitmap 数据, 才能取得后台画的 3D 对象, 并在可视的场景里, 把它显示到 Sprit3D 对象上。

```
buildingView.session=new BitmapSession(1);
```

使用 RectangleClipping 类是非常重要的, 因为它能减少 Sprite3D 对象画出的像素点。对后台画视口, 如果没有 RectangleClipping 工作, 画 3D 对象的空间是透明的。无论如何, 要完成将透明像素点画回到屏幕视口, 是要付出运行代价的。于是最好的办法就是限制后台画视口画出的像素点的数量。

```
buildingView.clipping=
    new RectangleClipping(
    {
        minx:-35,
        maxX:35,
        minY:-175,
```

```

        maxY:175
    }
};

```

幕后画视口还要有一个旋转照相机,因为当旋转照相机围绕屏幕视口旋转时,它也将围绕幕后画视口旋转,两个照相机的视角应该匹配。如此,如果 3D 对象直接添加到屏幕视口时,3D 对象的幕后画视口才接近于屏幕视口上看到的 3D 对象的情形。

```

buildingView. camera=
    new HoverCamera3D(
    {
        distance:1500,
        yfactor:1,
        tiltAngle:15
    }
    );

```

返回一个新的 buildingView,它能用 initObject()函数添加到 buildingViews 集合里。

这点是重要的,我们还没把这些幕后画视口添加到显示表里,这表明它们还是不可见的。

```

return buildingView;
}

```

为了移动旋转照相机,必须侦听一些鼠标事件,以下这些代码解释在第 7 章里。

```

protected override function initListeners():void
{
    super.initListeners();
    stage.addEventListener(
        MouseEvent.CLICK,
        mouseDown
    );
    stage.addEventListener(
        MouseEvent.CLICK,
        mouseUp
    );
    stage.addEventListener(
        MouseEvent.CLICK,
        mouseMove
    );
    protected override function initEnterFrame(event:Event):void
    {
        super.initEnterFrame(event);
    }
}

```

更新屏幕视口旋转照相机的位置:


```
(view.camera as HoverCamera3D).hover();
for each (var offscreenView:View3D in buildingViews)
{
```

每个幕后画视口的旋转照相机的位置也应更新到与屏幕视口旋转照相机的方向相匹配。

```
(offscreenView.camera as HoverCamera3D).hover();
```

还要画出每个幕后画视口：

```
    offscreenView.render();
}
}
protected override function initScene():void
{
    super.initScene();
```

每个建筑物 3D 对象,共享相同的材质,这里从嵌入的纹理来建立它。

```
var buildingMaterial:BitmapMaterial=
    new :BitmapMaterial(Cast.bitmap(BildingTexture));
```

然后,载入每个建筑物 3D 对象,从模型文件载入模型的内容在第 6 章里已介绍过。

```
var building1:ObjectContainer3D=
    Max3DS.parse(
        Building1,
        (
            autoLoadTextures:false,
            y:-200
        )
    );
var building2:ObjectContainer3D=
    Max3DS.parse(
        Building2,
        (
            autoLoadTextures:false,
            y:-200
        )
    );
var building3:ObjectContainer3D=
    Max3DS.parse(
        Building3,
        (
            autoLoadTextures:false,
            y:-200
```

```

    }
  );
var building4:ObjectContainer3D=
  Max3DS.parse(
    Building4,
    (
      autoLoadTextures:false,
      y:-200
    )
  );
var building5:ObjectContainer3D=
  Max3DS.parse(
    Building5,
    (
      autoLoadTextures:false,
      y:-200
    )
  );
for each (var container:ObjectContainer3D in[building1, building2, building3, building4, building5])
  for each (var child:Mesh in container.children)
    child.material=buildingMaterial;

```

然后,把每个建筑物 3D 对象添加到幕后画场景里。

```

buildingViews[0].scene.addChild(building1);
buildingViews[1].scene.addChild(building2);
buildingViews[2].scene.addChild(building3);
buildingViews[3].scene.addChild(building4);
buildingViews[4].scene.addChild(building5);

```

当 Sprite3D 对象添加到网格 Mesh 对象的时候,应用程序将在场景里显示所有 buildings 集合的对象。这里建立一个新的网格对象 Mesh,并把它添加到屏幕上的场景。

```

var sceneMesh:Mesh= new Mesh();
scene.addChild(sceneMesh);

```

在幕后画视口上循环 5 次:

```

var view:View3D;
var bitmap:BitmapData;
for (var i:int=0; i<5; ++i)
{

```

从 buildingViews 集合里得到每个幕后画视口:

```

view=buildingViews[i];

```

然后,引用 BitmapSession 画到视口里的 BitmapData 对象:

```
bitmap=(view.session as BitmapSession).getBitmapData(view);
```

最后,建立一个新的 BitmapMaterial 对象,这些材质要用于上面得到的 BitmapData 对象,现在把新的 BitmapMaterial 对象保存在 billboardMaterials 的集合里。

现在,当幕后画视口画一帧时,它会自动地使用相应的 BitmapMaterial 对象,这是因为视口 view 和 BitmapMaterial 两者都引用了相同的 BitmapData 对象。因此,使任意的 Sprite3D 对象把 BitmapMaterial 对象显示为材质。

```
billboardMaterials.push(  
    new BitmapMaterial(bitmap)  
);
```

把原始平面添加到场景,它代表建筑物地面,请注意,由于使用了 screenZOffset 初始化参数,这就确保地面总是画在包含建筑物的 Sprite3D 对象容器 Mesh 的下面。screenZOffset 初始化参数的介绍在第 4 章里。

```
scene.addChild(  
    new Plane(  
        {  
            Material:new BitmapMaterial(Cast.bitmap(GroundTexture)),  
            width:8000,  
            height:8000,  
            x:-66,  
            z: 66,  
            y:-130,  
            segments:20,  
            screenZOffset: 1000  
        }  
    )  
);
```

下面要完成的事是:建立 Sptite3D 对象,显示幕后画视口的输出,使用嵌套的 for 循环,在 x/z 平面的一个格里建立 1600 个布告板。

```
var randomMaterial:BitmapMaterial;  
var sprite:Sprite3D;  
for ( var xPos:int=-4000; xPos<4000; xPos+=200)  
{  
    for ( var zPos:int=-4000; zPos<4000; zPos+=200)  
    {
```

每个布告板显示一个随机的 billboardMaterial 集合里的 BitmapMaterial 对象。

```
randomMaterial=
    billboardMaterials[Math.round(Math.random()*4)];
```

然后,建立一个新的 Sprit3D 对象,提供随机的选用材质,再把它定位在此格子里。

```
sprite=new Sptite3D(randomMaterial);
sprite.x=xPos;
sprite.y=0;
sprite.z=zPos;
```

把 Sprit3D 对象添加到 Mesh 对象里,使它在场景里是可看见的。

```
sceneMesh.addSprite(sprite);
    }
}
```

mousedown()和 mouseUp()函数用于设置控制旋转照相机移动时的属性。

```
protected function mousedown(event:MouseEvent):void
{
    this.mouseButtonDown=true;
    this.lastStageX=event.stageX;
    this.lastStageY=event.stageY;
}
protected function mouseUp(event:MouseEvent):void
{
    this.mouseButtonDown=false;
}
```

为反映出最后帧上鼠标的运动,幕后画上的和屏幕上的旋转照相机的倾斜和摆动角度全都要更新。

```
protected function mouseMove(event:MouseEvent):void
{
    If (this.mouseButtonDown)
    {
        var pan:int=( event.stageX-lastStageX);
        var tilt:int=( event.stageY-lastStageY);
        (view.camera as HoverCamera3D).panAngle += pan;
        (view.camera as HoverCamera3D).tiltAngle += tilt;
        for each (var offscreenView:View3D in buildingViews)
        {
            (offscreenView.camera as HoverCamera3D).panAngle += pan;
            (offscreenView.camera as HoverCamera3D).tiltAngle += tilt;
        }
        this.lastStageX=event.stageX;
        this.lastStageY=event.stageY;
```



```
}  
}  
}  
}
```

最终的结果是一个 mesh 网格,带 1600 个 Sprit3D 对象元素,每个 Sprit3D 对象显示 5 个幕后画视口中其中一个的输出,而每个幕后画视口,画了一个建筑物 3D 对象。如果把每个建筑物 3D 对象都各个加到场景里,以合理的帧速画出来,这将是不可能的事情。

幕后画实际上完成了与 DirectionalSprite3D 类相同的结果(第 9 章中详细介绍了后者内容),使用幕后画比用 DirectionalSprite3D 类有以下三个好处。

(1) 幕后画所画出的模型,能从任意的角度看到,而用 DirectionalSprite3D 对象所画出的模型仅能从几个离散的角度可见。

(2) 如果 DirectionalSprite3D 对象从所有的视角画出几百个图像,它们的尺寸时常远超过单个屏幕 3D 模型和它的纹理的大小。

(3) 幕后画使用了一个单一的 3D 对象,它比 DirectionalSprite3D 对象预先画出所有图像,运行要更容易。

13.9 模型格式

Away3D 能装入各种模型格式,使用哪一种模型格式,要根据发布的应用程序的特性来确定,不同的模型格式,提供的特性不同。如果需要骨骼动画,可选用 Collada 格式,而 Quake2 MD2 格式,支持顶点动画。然而,对于静态模型,所有的格式都可使用。

这些模型格式的区别,对模型的负载和解析有重大的影响,此外还要考虑下面的这些因素:模型的复杂程度,动画,UV 数据,以及外部的材质。当选择模型格式的时候,不要只根据格式的功能的叙述,不同的格式能够对应用程序在载入时有重大的影响。在一定的环境下,一个模型格式载入的时间比动画快 10 倍。

如图 13-7 所示,表示用 Away3D 支持装入的 7 种不同模型格式装入同一个球模型时需要的平均时间值,以 3DS 作为比较的基准线。这个球是由 3970 个三角面组成的,不包含动画或骨骼,没有纹理,不含 UV 数据。这个模型球嵌入在 SWF 文件里,消除了载入外部文件时网络对它产生的各种影响。

正如所看到的,载入同样的一个 3D 模型所花费的时间,有相当大的不同,MD2、OBJ 和 3DS 格式相对快些,而 ASE、DAE、AWD 和 AS 格式要花更多时间。

现在看看对一个更复杂的球的载入时间,如图 13-8 所示。该球由 7922 个三角面组成(没有纹理、骨骼、动画或 UV 数据),再次地相对于 3DS 的其他格式的载入时间如图所示。因为 MD2 限制组成三角面的数只能是 4096,所以没有包含 MD2 格式。

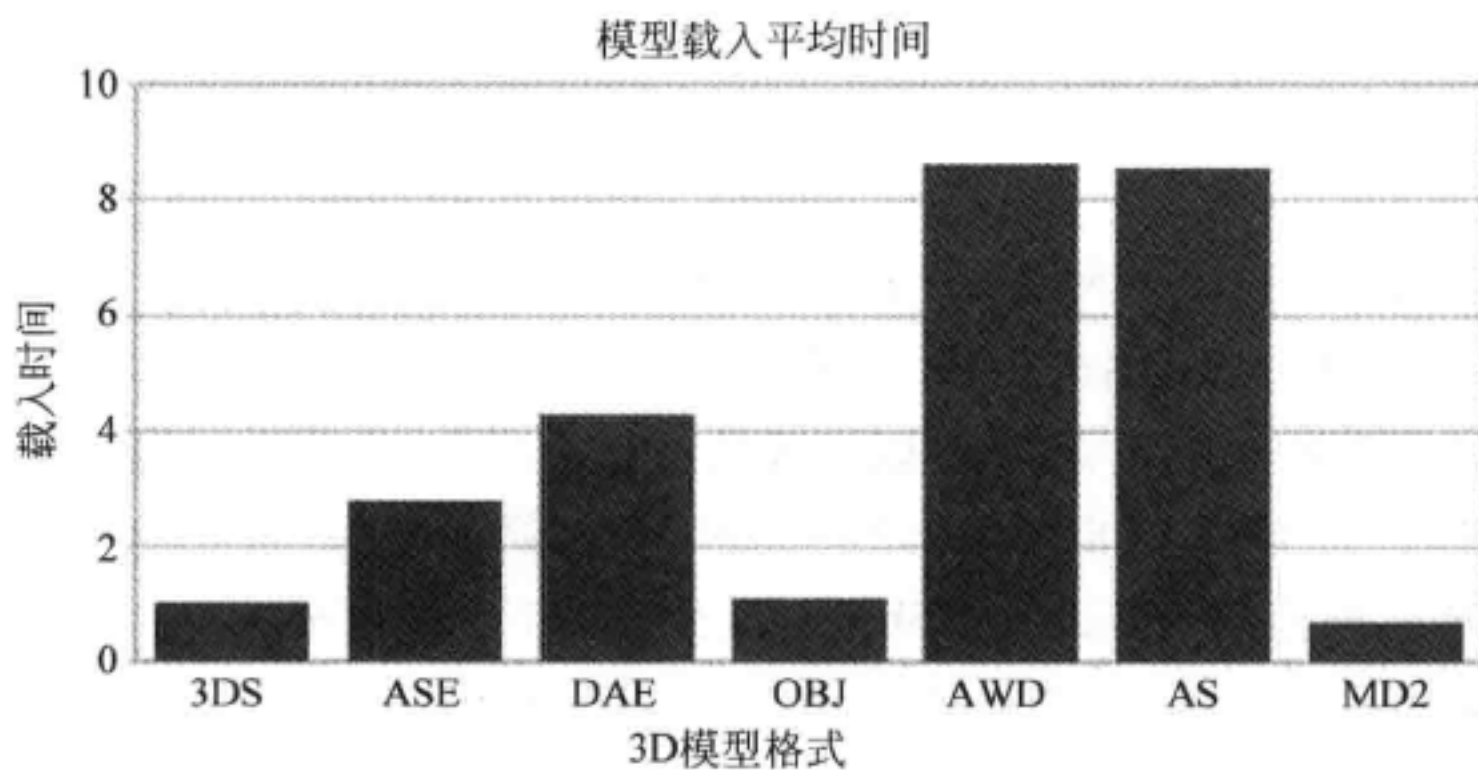


图 13-7 Away3D 支持文件格式及相应的导入时间

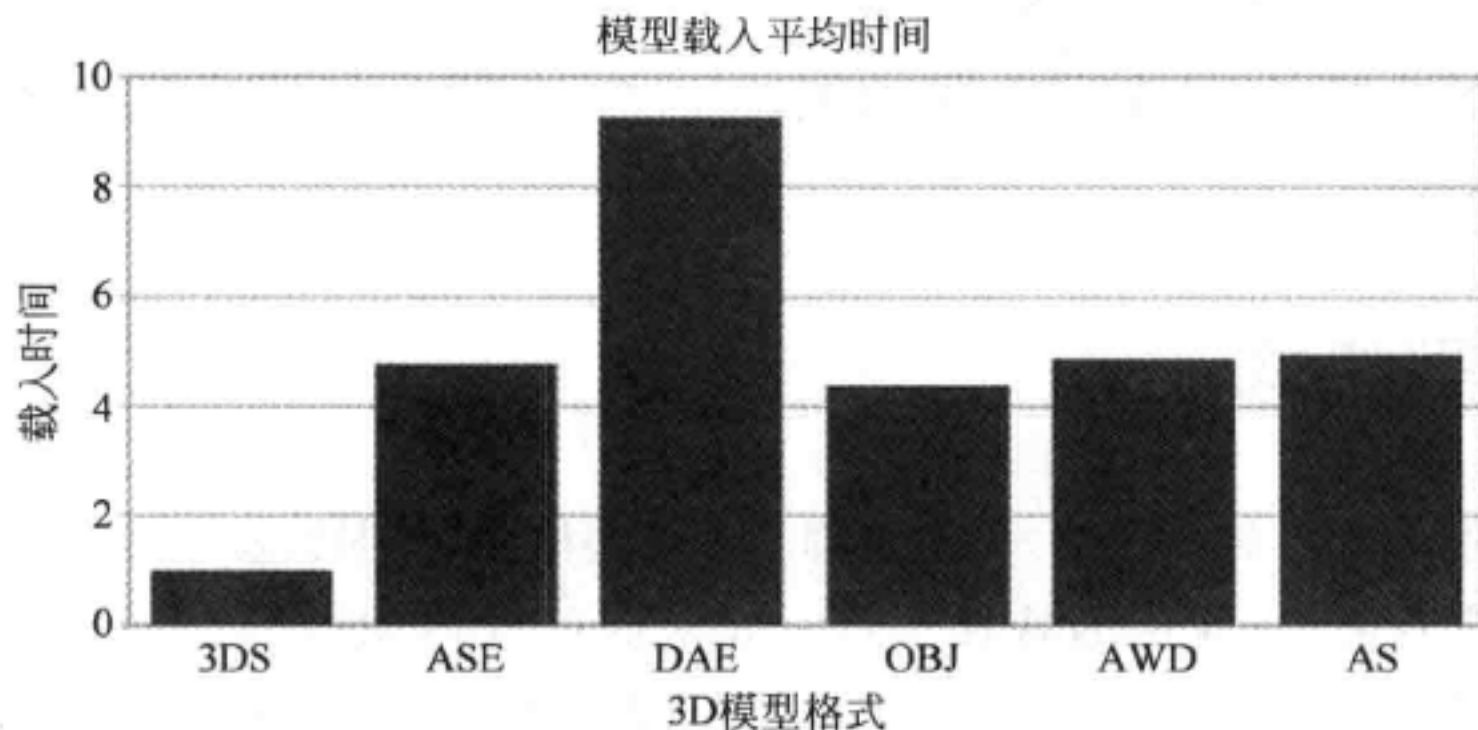


图 13-8 大量三角面构成的 3D 球导入时间比较

这里的情形又有些不同，很清楚，3DS 的载入时间最快。

从这些图表里得到的重要信息，不是它们本身这些特定的数值，而是一个事实：在不同的环境下，一些模型格式的载入时间比另外的格式要快很多，找到最好的模型格式，必须根据不同的情况来处理。如此做是值得的，因为对于载入 3D 模型文件，减少载入时间有巨大的潜力。

下面的应用程序 ModelLoadingSpeedTest 是一个例子，它可测量不同模型格式的载入时间，它以随机的顺序，5 次以不同的模型格式载入若干 3D 模型，然后在屏幕上显示平均的载入时间，很快地判断出哪种格式运行最快。

```
package
{
    import away3d.loaders.AWData;
    import away3d.loaders.Ase;
    import away3d.loaders.Collada;
```

```
import away3d.loaders.Max3DS;
import away3d.loaders.MD2;
import away3d.loaders.Obj;
import away3d.materials.WireColorMaterial;
import flash.events.Event;
import flash.text.TextField;
import flash.utils.getTimer;
public class ModelloadingSpeedTest extends Away3DTemplate
{
```

嵌入以各种格式保存的样本模型,当使用这些类测试 3D 模型的时候,必须更新 src 参数,反映出自己的 3D 模型文件的名字。

```
[Embed(source="TestSphere.3ds", mimeType="application/octet-stream")]
protected var TestModel3DS:Class;
[Embed(source="TestSphere.ase", mimeType="application/octet-stream")]
protected var TestModelASE:Class;
[Embed(source="TestSphere.dae", mimeType="application/octet-stream")]
protected var TestModelDAE:Class;
[Embed(source="TestSphere.obj", mimeType="application/octet-stream")]
protected var TestModelOBJ:Class;
[Embed(source="TestSphere.awd", mimeType="application/octet-stream")]
protected var TestModelAWD:Class;
[Embed(source="TestSphere.md2", mimeType="application/octet-stream")]
protected var TestModelMD2:Class;
```

下面的 7 个数组对象,用来保存在测试的时候,各种 3D 模型格式的载入时间。

```
protected var ThreeD3LoadTimes:Array=new Array();
protected var MD2LoadTimes:Array=new Array();
protected var ASELoadTimes:Array=new Array();
protected var DEALoadTimes:Array=new Array();
protected var OBJLoadTimes:Array=new Array();
protected var AWDLoadTimes:Array=new Array();
protected var ASLoadTimes:Array=new Array();
```

通过一个分开的函数,载入每个嵌入的 3D 模型,并记录载入过程的时间。这些函数将添加到 fuctionCalls 集合里,它让我们然后随机地选择下一个载入操作模型。

```
protected var functionCalls:Array=new Array();
```

TextField 对象由 results 属性来引用,用来显示应用程序的状态和当测试完成时的结果。

```
protected var results:TextField;
public function ModelloadingSpeedTest()
{
```

```
super();
```

把用来载入 3D 模型的 7 个函数添加到 functionCalls 集合里 5 次,这表示每个函数要调用 5 次,允许得到每个 3D 模型格式载入的平均时间。

```
for (var i:int=0; i<5; ++i)
{
    functionCalls.push(load3DS);
    functionCalls.push(loadASE);
    functionCalls.push(loadOBJ);
    functionCalls.push(loadDAE);
    functionCalls.push(loadAWD);
    functionCalls.push(loadAS);
    functionCalls.push(loadMD2);
}
protected override function initUI():void
{
    super.initUI();
    results=new TextField();
    results.x=10;
    results.y=10;
    results.width=300;
    addChild(results);
}
protected override function onEnterFrame(event:Event):void
{
    super.onEnterFrame(event);
```

在调用每一个函数的时候,该函数就要从 functionCalls 集合里删去。如果 functionCalls 集合里没有一个元素,就不必做测试过程。

```
if (functionCalls.length != 0)
{
```

屏幕上的状态文本要更新,显示还有多少剩下的函数要调用。

```
results.text=
    functionCalls.length + "model loading operations remaining";
```

在这里,需要从 functionCalls 集合数组里引用一个随机函数,并将它从集合数组里删除。

```
var randomPos:int=Math.round(
    Math.random() * (functionCalls.length-1));
var func:Function=functionCalls[randomPos];
functionCalls.splice(randomPos,1);
```


然后,调用该函数,并记录载入 3D 模型所花的时间。

```
func();
```

如果 functionCalls 数组集合里没有留下的函数,表示已运行完全部测试,现在能显示结果。

```
if (functionCalls.length == 0)
{
```

计算出每个不同的 3D 模型格式的平均载入时间:

```
var avg3DS:Number=getAverage(ThreeD3LoadTimes);
var avgASE:Number=getAverage(ASELoadTimes);
var avgDAE:Number=getAverage(DAELoadTimes);
var avgOBJ:Number=getAverage(OBJLoadTimes);
var avgAWD:Number=getAverage(AWDLoadTimes);
var avgAS:Number=getAverage(ASLoadTimes);
var avgMD2:Number=getAverage(MD2LoadTimes);
```

然后,将这个结果显示到屏幕上:

```
results.text=
    "MD2: " +
    avgMD2.toPrecision(4) +
    " seconds \n" +
    "3DS: " +
    avg3DS.toPrecision(4) +
    " seconds \n" +
    "ASE: " +
    avgASE.toPrecision(4) +
    " seconds \n" +
    "DAE: " +
    avgDAE.toPrecision(4) +
    " seconds \n" +
    "OBJ: " +
    avgOBJ.toPrecision(4) +
    " seconds \n" +
    "AWD: " +
    avgAWD.toPrecision(4) +
    " seconds \n" +
    "AS: " +
    avgAS.toPrecision(4) +
    " seconds "
}
```

getAverage()函数直接返回数组 Array 收集的载入一种模型格式所花时间的平均值。

```
protected function getAverage(array:Array):Number
{
    var average:Number=0;
    for each (var time:Number in array)
        average += time;
    return average/ array.length;
}
```

下面的 6 个函数,每个载入一种特定的 3D 模型格式。有关载入 3D 模型的问题,请参见第 6 章。

```
protected function loadMD2():void
{
```

记录 3D 模型载入前的当时时间,由 getTimer()函数返回的时间是毫秒数。

```
var start:int=getTimer();
```

载入 3D 模型:

```
Md2.parse(TestModelMD2);
```

记录 3D 模型载入后的当时时间:

```
var stop:int=getTimer();
```

计算载入 3D 模型所花的时间,单位是秒数:

```
var time:Number=(stop-start)/1000;
```

然后,把载入时间保存在相应的数组 Array 里:

```
MD2LoadTimes.push(time);
}
```

下面的各个函数使用了与上述 loadMD2()相同的设计逻辑,记录并保存载入给定的 3D 模型格式所花的时间。

```
protected function load3DS():void
{
    var start:int=getTimer();
    Max3DS.parse(
        TestModel3DS,
        {
            autoLoadTextures: false
        }
    )
}
```

```
    );  
    var stop:int=getTimer();  
    var time:Number=(stop-start)/1000;  
    ThreeD3LoadTimes.push(time);  
}  
protected function loadASE():void  
{  
    var start:int=getTimer();  
    Ase.parse(TestModelASE, {scaling: 1});  
    var stop:int=getTimer();  
    var time:Number=(stop-start)/1000;  
    ASELoadTimes.push(time);  
}  
protected function loadDAE():void  
{  
    var start:int=getTimer();  
    Collada.parse(TestModelDAE, );  
    var stop:int=getTimer();  
    var time:Number=(stop-start)/1000;  
    DAELoadTimes.push(time);  
}  
protected function loadOBJ():void  
{  
    var start:int=getTimer();  
    Obj.parse(  
        TestModelOBJ,  
        {  
            useMtl: false,  
            material:new WireColorMaterial()  
        }  
    );  
    var stop:int=getTimer();  
    var time:Number=(stop-start)/1000;  
    OBJLoadTimes.push(time);  
}  
protected function loadAWD():void  
{  
    var start:int=getTimer();  
    AWData.parse(TestModelAWD, { });  
    var stop:int=getTimer();  
    var time:Number=(stop-start)/1000;  
    AWDLoadTimes.push(time);  
}
```

```
protected function loadAS():void
{
    var start:int=getTimer();
    new TestSphere();
    var stop:int=getTimer();
    var time:Number=(stop-start)/1000;
    ASLoadTimes.push(time);
}
}
```